Engineering
Simplicity

Junos® OS

NETCONF XML Management Protocol Developer Guide

Published
2025-12-09

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

Table of Contents

About This Guide

Use this guide to remotely manage the configuration of devices running Junos OS using the Network Configuration Protocol (NETCONF), understand the native YANG data models on devices running Junos OS, or create YANG data models to add custom configuration hierarchies or RPCs to devices running Junos OS.

RELATED DOCUMENTATION

Junos XML Management Protocol Developer Guide

1

PART

Overview

CHAPTER 1

NETCONF XML Management Protocol Overview

IN THIS CHAPTER

Understanding the NETCONF XML Management Protocol

SUMMARY

Learn about the Network Configuration Protocol (NETCONF) and the advantages of using NETCONF to manage your network devices.

IN THIS SECTION

Benefits of NETCONF

- NETCONF is a standards-based protocol that was developed specifically for managing network devices.
- NETCONF uses secure, connection-oriented sessions that provide for authentication, data integrity, and confidentiality.
- NETCONF is vendor agnostic, so you can use the same NETCONF base operations to manage devices from different vendors.

NETCONF XML Management Protocol Overview

The NETCONF XML management protocol is a standards-based protocol that is specifically tailored for communicating with and managing network devices. NETCONF uses a client/server model and remote procedure call (RPC)-based communication. A NETCONF client establishes a connection and NETCONF session with a NETCONF server and executes operations on the device. Junos devices integrate the NETCONF server into the OS, and thus the server does not appear as a separate entry in process listings.

NETCONF uses XML-based data encoding for the RPCs and configuration data. The NETCONF protocol defines basic operations that are equivalent to CLI configuration mode commands. Client applications use the protocol operations to display, edit, and commit configuration data (among other operations), just as administrators use the CLI configuration mode commands to perform those operations. Within a NETCONF session, client applications can also execute RPCs equivalent to Junos OS operational mode commands.

Conceptually, NETCONF can be divided into 4 layers. Table 1 on page 3 describes the layers.

Table 1: NETCONF Layers

NETCONF Layer	Description
Transport	The transport layer facilitates the creation of sessions between the client and the server using various protocols.
Messages	The messages layer is a transport-independent framing mechanism for encoding RPCs and notifications. The tags include: • `<rpc>`—Encapsulates RPCs sent to the NETCONF server. • `<rpc-reply>`—Encapsulates RPC replies received from the NETCONF server, which can include data, `<ok/>` tags, and errors and warnings. • `<notification>`—Encapsulates notification messages, which are one-way messages that the NETCONF server sends asynchronously to NETCONF clients that subscribe to notifications. For more information about NETCONF event notifications, see "NETCONF Event Notifications" on page 165.
Operations	The operations layer defines the protocol operations that you can perform on a network device. The operations comprise base protocol operations, for example, `<get-config>`, `<edit-config>`, and `<commit>`, as well as vendor-specific operations. A client application invokes the operations as RPCs and provides XML-encoded parameters. See the *NETCONF Protocol Operations and Attributes* section of this guide for more information about specific operations.

Table 1: NETCONF Layers *(Continued)*

NETCONF Layer	Description
Content	The content layer contains the RPC request and response payloads in XML format. This layer defines the configuration data and the notification data.

Communication between the NETCONF server and a client application is session based. The server and client explicitly establish a connection and session before exchanging data. As defined by the transport layer, NETCONF can use any secure transport protocol that meets the necessary requirements. Junos devices support NETCONF sessions over SSH, outbound SSH, TLS, and outbound HTTPS, as well as NETCONF Call Home sessions over outbound SSH.

Each NETCONF session begins with a handshake, in which the server and client exchange `<hello>` tags that enclose their supported NETCONF capabilities. Within a NETCONF session, the client and server exchange messages, which contain RPCs, RPC replies, or notifications. The NETCONF operations layer defines the protocol operations that a client application can use to manage a device. The content layer describes the encoded parameters and data included in the RPCs. "NETCONF and the Junos XML API Overview" on page 4 describes the content layer in more detail. After the client application finishes making requests, it closes the NETCONF session and connection.

A NETCONF client sends RPCs to the NETCONF server to request information, execute operational commands, or modify the configuration on a device. The NETCONF server processes the RPCs and sends RPC replies to the client. Depending on the request, RPC replies return requested information or indicate the success or failure of the requested operation.

For more information about NETCONF, see the following RFCS:

- RFC 6241, *Network Configuration Protocol (NETCONF)*
- RFC 6242, *Using the NETCONF Protocol over Secure Shell (SSH)*

NETCONF and the Junos XML API Overview

The Junos XML API is an XML representation of Junos OS configuration statements and operational mode commands. The Junos XML API defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the commands that you issue in CLI operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element. Junos XML request tags are equivalent in function to the corresponding operational mode commands in the CLI.

NETCONF client applications can request information, execute operational commands, or modify the configuration on a Junos device. The client application encodes the request in NETCONF and Junos XML API tag elements and sends it to the NETCONF server on the device. The NETCONF server directs the request to the appropriate software modules, encodes the response in NETCONF and Junos XML API tag elements, and returns the result to the client application.

When a NETCONF client modifies the Junos OS configuration, the RPC content includes Junos XML configuration data. NETCONF clients can also send operational RPCs with the appropriate request tags to execute operational commands or retrieve information. The server returns the response using Junos XML API elements enclosed within the corresponding response tag element.

For example, to request information about the status of a device's interfaces, a client application sends the Junos XML API `<get-interface-information/>` request tag.

```
<rpc>
    <get-interface-information/>
</rpc>
```

The NETCONF server gathers the information from the interface process and returns it in the Junos XML API `<interface-information>` response tag element.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/25.2R1/junos">
<interface-information xmlns="http://xml.juniper.net/junos/25.2R1/junos-interface"
junos:style="normal">
<physical-interface>
<name>
ge-0/0/0
</name>
<admin-status junos:format="Enabled">
up
</admin-status>
...
</interface-information>
</rpc-reply>
```

You can determine Junos XML API content in a number of ways. The Juniper Networks XML API Explorer enables you to browse Junos XML API elements. You can view the configuration elements and the operational request and response tags supported in a given OS and release.

Additionally, on Junos devices, you can use the pipe (|) operator in the CLI to view Junos XML API content. For example, to retrieve the operational request tag for a given command, issue *`command`* `| display` `xml rpc` in the CLI. The following example shows that the request tag for the `show interfaces` command is `<get-interface-information>`.

```
user@host> show interfaces | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1/junos">
    <rpc>
        <get-interface-information>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Similarly, to retrieve configuration data in XML format use the `show configuration | display xml` command.

```
user@host> show configuration system services | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1/junos">
    <configuration junos:commit-seconds="1747272887" junos:commit-localtime="2025-05-14 18:34:47
PDT" junos:commit-user="admin">
        <system>
            <services>
                <netconf>
                    <ssh>
                    </ssh>
                </netconf>
                <ssh>
                    <root-login>allow</root-login>
                </ssh>
                <ftp>
                </ftp>
            </services>
        </system>
    </configuration>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

You can use NETCONF and the Junos XML API to manage Junos devices. You can write client applications to interact with the NETCONF server. You can also use NETCONF to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

Advantages of Using NETCONF and the Junos XML API

IN THIS SECTION

NETCONF and the Junos XML API fully document all options for every supported Junos OS operational request and all elements in every Junos OS *configuration statement*. The tag names clearly indicate the function of an element in an operational request or configuration statement.

The combination of meaningful tag names and the structural rules in a DTD makes it easy to understand the content and structure of an XML-tagged data set. NETCONF and Junos XML tag elements make it straightforward for client applications to parse the device output to find and display specific information, as described in the following sections.

Parse Device Output

The following example illustrates how the Junos XML API makes it easier to parse device output and extract the needed information. The example compares formatted ASCII and XML-tagged output from a device running Junos OS. The formatted ASCII output is:

```
Physical interface: fxp0, Enabled, Physical link is Up
   Interface index: 64, SNMP ifIndex: 1
```

The corresponding XML-tagged version is:

```
<physical-interface>
    <name>fxp0</name>
    <admin-status junos:format="Enabled">up</admin-status>
    <oper-status>up</oper-status>
    <local-index>64</local-index>
    <snmp-index>1</snmp-index>
    ...
</physical-interface>
```

When a client application needs to extract a specific value from formatted ASCII output, it must rely on the value's location, expressed either absolutely or with respect to labels or values in adjacent fields. Suppose that the client application wants to extract the interface index. It can use a regular-expression matching utility to locate specific strings, but the number of digits in the interface index is not necessarily predictable. The client application cannot simply read a certain number of characters after the `Interface index:` label. Instead, it must extract everything between the label and the subsequent label, which is:

```
, SNMP ifIndex
```

A problem arises if the format or ordering of output changes in a later version of the Junos OS. For example, the output might add a `Logical index` field following the interface index number.

```
Physical interface: fxp0, Enabled, Physical link is Up
    Interface index: 64, Logical index: 12, SNMP ifIndex: 1
```

An application that extracts the interface index number delimited by the `Interface index:` and `SNMP ifIndex` labels now obtains an incorrect result. In this case, you must update the application to manually search for the following label instead:

```
, Logical index
```

In contrast, the structured nature of XML-tagged output enables a client application to retrieve the interface index by extracting everything within the opening `<local-index>` tag and closing `</local-index>` tag. The application does not have to rely on an element's position in the output string. As a result, the NETCONF server can emit the child tag elements in any order within the `<physical-interface>` element. Adding a new `<logical-index>` element does not affect an application's ability to locate the `<local-index>` element and extract its contents.

Display Device Output

XML-tagged output is also easier to transform into different display formats. For example, you might want to display different amounts of detail about a given device component at different times. When a device returns formatted ASCII output, you must write special routines and data structures in your application to extract the information needed for a given detail level. In contrast, the inherent structure of XML output is an ideal basis for a display program's own structures. You can use the same extraction routine for several levels of detail, simply ignoring the tag elements that you don't need when displaying less detail.

RELATED DOCUMENTATION

XML and Junos OS Overview

XML Overview

CHAPTER 2

NETCONF and Junos XML Tags Overview

IN THIS CHAPTER

XML and Junos OS Overview

Extensible Markup Language (XML) is a standard for representing and communicating information. It is a metalanguage for defining customized tags that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. Junos OS natively supports XML for the operation and configuration of devices running Junos OS.

The Junos OS *command-line interface* (*CLI*) and the Junos OS infrastructure communicate using XML. When you issue an *operational mode command* in the CLI, the CLI converts the command into XML format for processing. After processing, Junos OS returns the output in the form of an XML document, which the CLI converts back into a readable format for display. Remote client applications also use XML-based data encoding for operational and configuration requests on devices running Junos OS.

The Junos XML *API* is an XML representation of Junos OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the commands that you issue in CLI operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element.

To display the configuration or operational mode command output as Junos XML tag elements instead of as the default formatted ASCII, issue the command, and pipe the output to the `display xml` command. Infrastructure tag elements in the response belong to the Junos XML management protocol. The tag elements that describe Junos OS configuration or operational data belong to the Junos XML API, which defines the Junos OS content that can be retrieved and manipulated by both the Junos XML management protocol and the NETCONF XML management protocol operations. The following example compares the text and XML output for the `show chassis alarms` operational mode command:

```
user@host> show chassis alarms
No alarms currently active
```

```
user@host> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/10.4R1/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

To display the Junos XML API representation of any operational mode command, issue the command, and pipe the output to the `display xml rpc` command. The following example shows the Junos XML API request tag for the `show chassis alarms` command.

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
    <rpc>
        <get-alarm-information>
        </get-alarm-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

As shown in the previous example, the `| display xml rpc` option displays the Junos XML API request tag that is sent to Junos OS for processing whenever the command is issued. In contrast, the `| display xml` option displays the actual output of the processed command in XML format.

When you issue the `show chassis alarms` operational mode command, the CLI converts the command into the Junos XML API `<get-alarm-information>` request tag and sends the XML request to the Junos OS infrastructure for processing. Junos OS processes the request and returns the `<alarm-information>` response tag element to the CLI. The CLI then converts the XML output into the “No alarms currently active” message that is displayed to the user.

Junos OS automation scripts use XML to communicate with the host device. Junos OS provides XML-formatted input to a script. The script processes the input source tree and then returns XML-formatted output to Junos OS. The script type determines the XML input document that is sent to the script as well as the output document that is returned to Junos OS for processing. Commit script input consists of an XML representation of the post-inheritance candidate configuration file. Event scripts receive an XML document containing the description of the triggering event. All script input documents contain information pertaining to the Junos OS environment, and some scripts receive additional script-specific input that depends on the script type.

RELATED DOCUMENTATION

Junos XML API Explorer

XML Overview

IN THIS SECTION

Extensible Markup Language (XML) is a language for defining a set of markers, called *tags*, that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. XML tags look much like HTML tags, but XML is actually a metalanguage used to define tags that best suit the kind of data being marked.

For more details about XML, see *A Technical Introduction to XML* at http://www.xml.com/pub/a/98/10/guide0.html and the additional reference material at the http://www.xml.com site. The official XML specification from the World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, is available at http://www.w3.org/TR/REC-xml.

The following sections discuss general aspects of XML.

Tag Elements

XML has three types of tags: opening tags, closing tags, and empty tags. XML tag names are enclosed in angle brackets and are case sensitive. Items in an XML-compliant document or data set are always enclosed in paired opening and closing tags, and the tags must be properly nested. That is, you must close the tags in the same order in which you opened them. XML is stricter in this respect than HTML, which sometimes uses only opening tags. The following examples show paired opening and closing tags enclosing a value. The closing tags are indicated by the forward slash at the start of the tag name.

```
<interface-state>enabled</interface-state>
<input-bytes>25378</input-bytes>
```

The term *tag element* or *element* refers to a three-part set: opening tag, contents, and closing tag. The content can be an alphanumeric character string as in the preceding examples, or can itself be a *container* tag element, which contains other tag elements. For simplicity, the term *tag* is often used interchangeably with *tag element* or *element*.

If an element is *empty*—has no contents—it can be represented either as paired opening and closing tags with nothing between them, or as a single tag with a forward slash after the tag name. For example, the notation `<snmp-trap-flag/>` is equivalent to `<snmp-trap-flag></snmp-trap-flag>`.

As the preceding examples show, angle brackets enclose the name of the element. This is an XML convention, and the brackets are a required part of the complete element name. They are not to be confused with the angle brackets used in the Juniper Networks documentation to indicate optional parts of Junos OS CLI command strings.

Junos XML elements follow the XML convention that the element name indicates the kind of information enclosed by the tags. For example, the Junos XML `<interface-state>` element indicates that it contains a description of the current status of an interface on the device, whereas the `<input-bytes>` element indicates that its contents specify the number of bytes received.

When discussing XML elements in text, this documentation conventionally uses just the opening tag to represent the complete element (opening tag, contents, and closing tag). For example, the documentation refers to the `<input-bytes>` tag to indicate the entire `<input-bytes>`*`number-of-bytes`*`</input-bytes>` element.

Attributes

XML elements can contain associated properties in the form of *attributes*, which specify additional information about an element. Attributes appear in the opening tag of an element and consist of an attribute name and value pair. The attribute syntax consists of the attribute name followed by an equals sign and then the attribute value enclosed in quotation marks. An XML element can have multiple attributes. Multiple attributes are separated by spaces and can appear in any order.

In the following example, the `configuration` element has two attributes, `junos:changed-seconds` and `junos:changed-localtime`.

```
<configuration junos:changed-seconds="1279908006" junos:changed-localtime="2010-07-23 11:00:06
PDT">
```

The value of the `junos:changed-seconds` attribute is "1279908006", and the value of the `junos:changed-localtime` attribute is "2010-07-23 11:00:06 PDT".

Namespaces

Namespaces allow an XML document to contain the same tag, attribute, or function names for different purposes and avoid name conflicts. For example, many namespaces may define a `print` function, and each may exhibit a different functionality. To use the functionality defined in one specific namespace, you must associate that function with the namespace that defines the desired functionality.

To refer to a tag, attribute, or function from a defined namespace, you must first provide the namespace *Uniform Resource Identifier* (URI) in your style sheet declaration . You then qualify a tag, attribute, or function from the namespace with the URI. Since a URI is often lengthy, generally a shorter prefix is mapped to the URI.

In the following example the `jcs` prefix is mapped to the namespace identified by the URI `http://xml.juniper.net/junos/commit-scripts/1.0`, which defines extension functions used in commit, op, event, and SNMP scripts. The `jcs` prefix is then prepended to the `output` function, which is defined in that namespace.

```
<?xml version="1.0"?>
 <xsl:stylesheet version="1.0" xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
    ...
    <xsl:value-of select="jcs:output('The VPN is up.')"/>
 </xsl: stylesheet>
```

During processing, the prefix is expanded into the URI reference. Although there may be multiple namespaces that define an `output` element or function, the use of `jcs:output` explicitly defines which `output` function is used. You can choose any prefix to refer to the contents in a namespace, but there must be an existing declaration in the XML document that binds the prefix to the associated URI.

Document Type Definition

An XML-tagged document or data set is *structured* because a set of rules specifies the ordering and interrelationships of the items in it. A file called a *document type definition*, or *DTD*, defines these rules. The rules define the contexts in which each tagged item can—and in some cases must—occur. A DTD:

- Lists every element that can appear in the document or data set
- Defines the parent-child relationships between the tags
- Specifies other tag characteristics

The same DTD can apply to many XML documents or data sets.

RELATED DOCUMENTATION

Junos XML Management Protocol and Junos XML API Overview

XML and Junos OS Overview

XML and NETCONF XML Management Protocol Conventions Overview

IN THIS SECTION

A client application must comply with XML and NETCONF XML management protocol conventions. Each request from the client application must be a *well-formed* XML document; that is, it must obey the structural rules defined in the NETCONF and Junos XML document type definitions (DTD)s for the kind of information encoded in the request. The client application must emit tag elements in the required order and only in the legal contexts. Compliant applications are easier to maintain in the event of changes to the Junos OS or NETCONF protocol.

Similarly, each response from the NETCONF server constitutes a well-formed XML document (the NETCONF server obeys XML and NETCONF conventions).

The following sections describe NETCONF XML management protocol conventions:

Request and Response Tag Elements

A *request* tag element is one generated by a client application to request information about a device's current status or configuration, or to change the configuration. A request tag element corresponds to a CLI operational or configuration command. It can occur only within an `<rpc>` tag.

A *response* tag element represents the NETCONF server's reply to a request tag element and occurs only within an `<rpc-reply>` tag.

The following example represents an exchange in which a client application emits the `<get-interface-information>` request tag element with the `<extensive/>` flag and the NETCONF server returns the `<interface-information>` response tag element.

Client Application

```
<rpc>
    <get-interface-information>
        <extensive/>
    </get-interface-information>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <interface-information xmlns="URL">
        <!-- children of <interface-information> -->
    </interface-information>
</rpc-reply>
]]>]]>
```

NOTE: This example, like all others in this guide, shows each tag element on a separate line, in the tag streams emitted by both the client application and NETCONF server. In practice, a client application does not need to include newline characters between tag elements, because the server automatically discards such white space. For further discussion, see "Spaces, Newline Characters, and Other White Space" on page 18.

Child Tag Elements of a Request Tag Element

Some request tag elements contain child tag elements. For configuration requests, each child tag element represents a configuration element (hierarchy level or configuration object). For operational requests, each child tag element represents one of the options you provide on the command line when issuing the equivalent CLI command.

Some requests have mandatory child tag elements. To make a request successfully, a client application must emit the mandatory tag elements within the request tag element's opening and closing tags. If any of the children are themselves container tag elements, the opening tag for each must occur before any of the tag elements it contains, and the closing tag must occur before the opening tag for another tag element at its hierarchy level.

In most cases, the client application can emit children that occur at the same level within a container tag element in any order. The important exception is a configuration element that has an *identifier tag element*, which distinguishes the configuration element from other elements of its type. The identifier tag element must be the first child tag element in the container tag element. Most frequently, the identifier tag element specifies the name of the configuration element and is called `<name>`. For more information, see "Mapping for Objects That Have an Identifier" on page 26.

Child Tag Elements of a Response Tag Element

The child tag elements of a response tag element represent the individual data items returned by the NETCONF server for a particular request. The children can be either individual tag elements (empty tags or tag element triples) or container tag elements that enclose their own child tag elements. For some container tag elements, the NETCONF server returns the children in alphabetical order. For other elements, the children appear in the order in which they were created in the configuration.

The set of child tag elements that can occur in a response or within a container tag element is subject to change in later releases of the Junos XML API. Client applications must not rely on the presence or absence of a particular tag element in the NETCONF server's output, nor on the ordering of child tag elements within a response tag element. For the most robust operation, include logic in the client application that handles the absence of expected tag elements or the presence of unexpected ones as gracefully as possible.

Spaces, Newline Characters, and Other White Space

As dictated by the XML specification, the NETCONF server ignores white space (spaces, tabs, newline characters, and other characters that represent white space) that occurs between tag elements in the tag stream generated by a client application. Client applications can, but do not need to, include white space between tag elements. However, they must not insert white space within an opening or closing tag. If they include white space in the contents of a tag element that they are submitting as a change to the candidate configuration, the NETCONF server preserves the white space in the configuration database.

In its responses, the NETCONF server includes white space between tag elements to enhance the readability of responses that are saved to a file: it uses newline characters to put each tag element on its own line, and spaces to indent child tag elements to the right compared to their parents. A client application can ignore or discard the white space, particularly if it does not store responses for later review by human users. However, it must not depend on the presence or absence of white space in any particular location when parsing the tag stream.

For more information about white space in XML documents, see the XML specification from the World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, at http://www.w3.org/TR/REC-xml/ .

XML Comments

Client applications and the NETCONF server can insert XML comments at any point between tag elements in the tag stream they generate, but not within tag elements. Client applications must handle comments in output from the NETCONF server gracefully but must not depend on their content. Client applications also cannot use comments to convey information to the NETCONF server, because the server automatically discards any comments it receives.

XML comments are enclosed within the strings <!-- and -->, and cannot contain the string -- (two hyphens). For more details about comments, see the XML specification at http://www.w3.org/TR/REC-xml/ .

The following is an example of an XML comment:

```
<!-- This is a comment. Please ignore it. -->
```

Predefined Entity References

By XML convention, there are two contexts in which certain characters cannot appear in their regular form:

- In the string that appears between opening and closing tags (the contents of the tag element)

- In the string value assigned to an attribute of an opening tag

When including a disallowed character in either context, client applications must substitute the equivalent *predefined entity reference*, which is a string of characters that represents the disallowed character. Because the NETCONF server uses the same predefined entity references in its response tag elements, the client application must be able to convert them to actual characters when processing response tag elements.

Table 2 on page 19 summarizes the mapping between disallowed characters and predefined entity references for strings that appear between the opening and closing tags of a tag element.

Table 2: Predefined Entity Reference Substitutions for Tag Content Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&amp;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;

Table 3 on page 19 summarizes the mapping between disallowed characters and predefined entity references for attribute values.

Table 3: Predefined Entity Reference Substitutions for Attribute Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&amp;
' (apostrophe)	&apos;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;
" (quotation mark)	&quot;

As an example, suppose that the following string is the value contained by the `<condition>` tag element:

```
if (a<b && b>c) return "Peer’s not responding"
```

The `<condition>` tag element looks like this (it appears on two lines for legibility only):

```
<condition>if (a&lt;b &amp;&amp; b&gt;c) return "Peer’s not \
    responding"</condition>
```

Similarly, if the value for the `<example>` tag element’s `heading` attribute is `Peer’s "age" <> 40`, the opening tag looks like this:

```
<example heading="Peer&apos;s &quot;age&quot; &lt;&gt; 40">
```

Map Junos OS Commands and Command Output to Junos XML Tag Elements

IN THIS SECTION

The Junos XML API is an XML representation of Junos OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the commands that you issue in CLI operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element.

Request tag elements are used in remote procedure calls (RPCs) within NETCONF and Junos XML protocol sessions to request information from a device running Junos OS or a device running Junos OS Evolved. The server returns the response using Junos XML tag elements enclosed within the response tag element. For example, the `show interfaces` command maps to the `<get-interface-information>` request tag, and the server returns the `<interface-information>` response tag.

The following sections outline how to map commands, command options, and command output to Junos XML tag elements.

Mapping Command Output to Junos XML Elements

On the Junos OS CLI, to display command output as Junos XML elements instead of as the default formatted ASCII text, include the `| display xml` option after the command. The XML elements that describe the Junos OS configuration or operational data belong to the Junos XML API. The Junos XML API defines the Junos OS content that can be retrieved and manipulated by NETCONF and Junos XML management protocol operations.

The following example shows the output from the `show chassis hardware` command. The output is identical to the server's response for the `<get-chassis-inventory>` RPC request.

```
user@host> show chassis hardware | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <chassis-inventory xmlns="http://xml.juniper.net/junos/25.2R0/junos-chassis">
        <chassis junos:style="inventory">
            <name>Chassis</name>
            <serial-number>JN1085AA1AFA</serial-number>
            <description>MX960</description>
            <chassis-module>
                <name>Midplane</name>
                <version>REV 02</version>
                <part-number>710-013698</part-number>
                <serial-number>AA0001</serial-number>
                <description>MX960 Backplane</description>
                <model-number>CHAS-BP-MX960-S</model-number>
            </chassis-module>
            <chassis-module>
                <name>FPM Board</name>
                <version>REV 02</version>
                <part-number>710-014974</part-number>
                <serial-number>AA0002</serial-number>
                <description>Front Panel Display</description>
                <model-number>CRAFT-MX960-S</model-number>
            </chassis-module>
            <chassis-module>
                <name>PDM</name>
                <version>Rev 02</version>
                <part-number>740-013110</part-number>
                <serial-number>AAA0000001A</serial-number>
                <description>Power Distribution Module</description>
            </chassis-module>
            <!-- other child tags of <chassis> -->
        </chassis>
    </chassis-inventory>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Mapping Commands to Junos XML Request Tag Elements

You can find information about the available Junos OS or Junos OS Evolved operational mode commands and their equivalent Junos XML RPC request tags using the following methods:

- Appending `| display xml rpc` to an operational command in the CLI.
- Using the Junos XML API Explorer - Operational Tags application to search for a command or request tag in a given release.

You can use the Junos XML API Explorer tool to: verify a command, map the command to its equivalent Junos XML RPC request tag and child tags, and view the expected response tag for various Junos OS or Junos OS Evolved releases.

In the Junos OS CLI, you can display the Junos XML request tag elements for any operational mode command that has a Junos XML counterpart. To display the request tags for a given command, enter the command and pipe it to the `display xml rpc` command.

The following example displays the RPC tags for the `show route` command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE: Starting in Junos OS Release 20.3R1, the names of some Junos XML RPC request tags have been updated to ensure consistency across the Junos XML API. Junos devices still accept the old request tag names for backwards compatibility, but we recommend using the new names going forward. To verify the Junos XML RPC request tag for an operational mode command in a given Junos OS release, see the Junos XML API Explorer - Operational Tags tool.

Mapping for Command Options with Variable Values

Many CLI commands have options that identify the object that the command affects or reports on, distinguishing the object from other objects of the same type. In some cases, the CLI does not precede the identifier with a fixed-form keyword, but XML convention requires that the Junos XML API define a tag element for every option. To find the names for each identifier (and any other child tag elements) for an operational request tag element, consult the tag element's entry in the appropriate DTD. Alternatively, issue the command and command option in the CLI and append the `| display xml rpc` option.

Table 4 on page 23 shows the Junos XML tag elements for two operational commands that have variable-form options. In the `show interfaces` command, ge-0/0/1 is the name of the interface. In the `show bgp neighbor` command, 10.168.1.222 is the IP address for the BGP peer of interest.

Table 4: Commands with Variable-Form Options

Command	Junos XML Tags

`show interfaces ge-0/0/1`	`<rpc>`
	`  <get-interface-information>`
	`    <interface-name>ge-0/0/1</interface-name>`
	`  </get-interface-information>`
	`</rpc>`

`show bgp neighbor 10.168.1.122`	`<rpc>`
	`  <get-bgp-neighbor-information>`
	`    <neighbor-address>10.168.1.122</neighbor-address>`
	`  </get-bgp-neighbor-information>`
	`</rpc>`

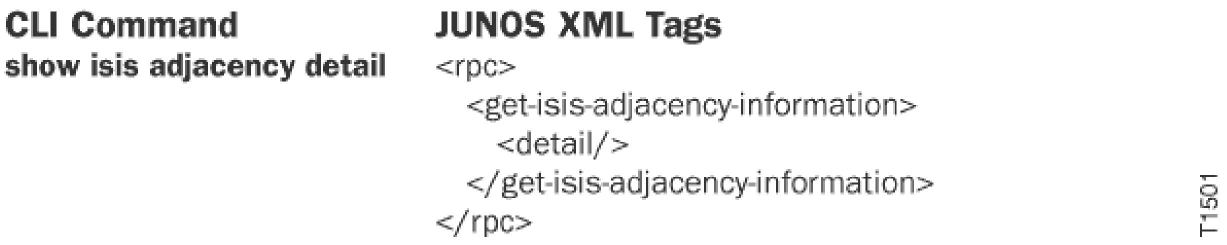
You can display the Junos XML RPC tags for a command and its options in the CLI by executing the command and command option and appending `| display xml rpc`.

```
user@host> show interfaces ge-0/0/1 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <rpc>
        <get-interface-information>
            <interface-name>ge-0/0/1</interface-name>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Mapping for Fixed-Form Command Options

Some CLI commands include options that have a fixed form, such as the `brief` and `detail` options, which specify the amount of detail to include in the output. The Junos XML API usually maps such an option to an empty tag whose name matches the option name.

The following example shows the Junos XML tag elements for the `show isis adjacency` command, which has a fixed-form option called `detail`:



CLI Command
show isis adjacency detail

JUNOS XML Tags

```
<rpc>
  <get-isis-adjacency-information>
    <detail/>
  </get-isis-adjacency-information>
</rpc>
```

To view the tags in the CLI:

```
user@host> show isis adjacency detail | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <rpc>
        <get-isis-adjacency-information>
            <detail/>
        </get-isis-adjacency-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
20.3R1	Starting in Junos OS Release 20.3R1, the names of some Junos XML RPC request tags have been updated to ensure consistency across the Junos XML API.

Map Configuration Statements to Junos XML Tag Elements

IN THIS SECTION

The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy. At the top levels of the configuration hierarchy, there is almost always a one-to-one mapping between tag elements and statements, and most tag names match the configuration statement name. At deeper levels of the hierarchy, the mapping is sometimes less direct, because some CLI notational conventions do not map directly to XML-compliant tagging syntax.

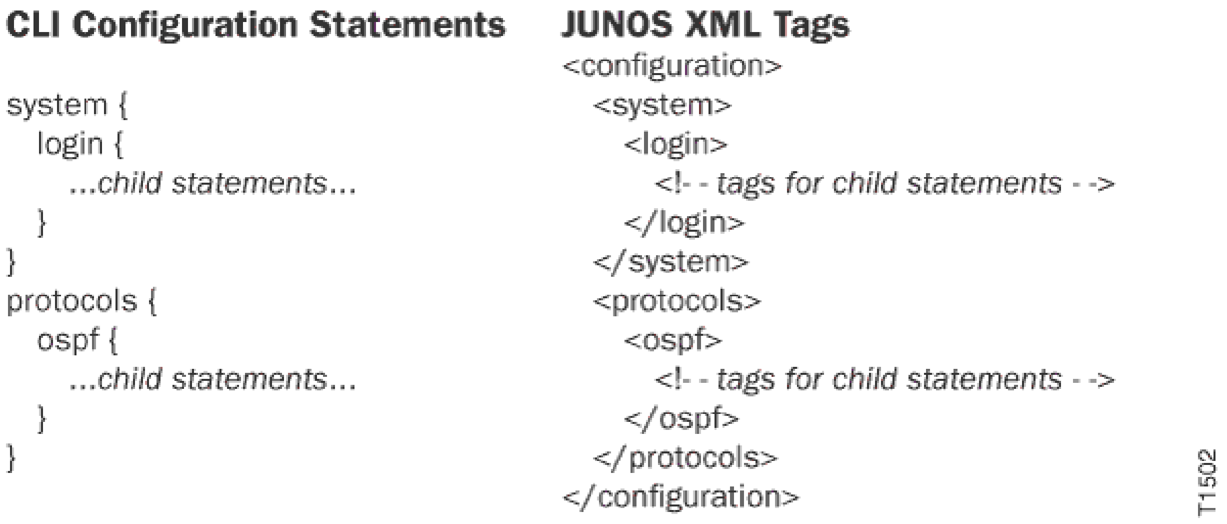
NOTE: For some configuration statements, the notation used when you type the statement at the CLI configuration-mode prompt differs from the notation used in a configuration file. The same Junos XML tag element maps to both notational styles.

The following sections describe the mapping between configuration statements and Junos XML tag elements:

Mapping for Hierarchy Levels and Container Statements

The `<configuration>` element is the top-level Junos XML container element for configuration statements. It corresponds to the `[edit]` hierarchy level in CLI configuration mode. Most statements at the next few levels of the configuration hierarchy are container statements. The Junos XML container tag element that corresponds to a container statement almost always has the same name as the statement.

The following example shows the Junos XML tag elements for two statements at the top level of the configuration hierarchy. Note that a closing brace in a CLI configuration statement corresponds to a closing Junos XML tag.



CLI Configuration Statements

```
system {
   login {
      ...child statements...
   }
}
protocols {
   ospf {
      ...child statements...
   }
}
```

JUNOS XML Tags

```
<configuration>
   <system>
      <login>
         <!- - tags for child statements - ->
      </login>
   </system>
   <protocols>
      <ospf>
         <!- - tags for child statements - ->
      </ospf>
   </protocols>
</configuration>
```

Mapping for Objects That Have an Identifier

At some hierarchy levels, the same kind of configuration object can occur multiple times. Each instance of the object has a unique identifier to distinguish it from the other instances. In the CLI notation, the parent statement for such an object consists of a keyword and identifier of the following form:

```
keyword identifier {
… configuration statements for individual characteristics …
}
```

`keyword` is a fixed string that indicates the type of object being defined, and *`identifier`* is the unique name for this instance of the type. In the Junos XML API, the tag element corresponding to the keyword is a container tag element for child tag elements that represent the object's characteristics. The container tag element's name generally matches the `keyword` string.

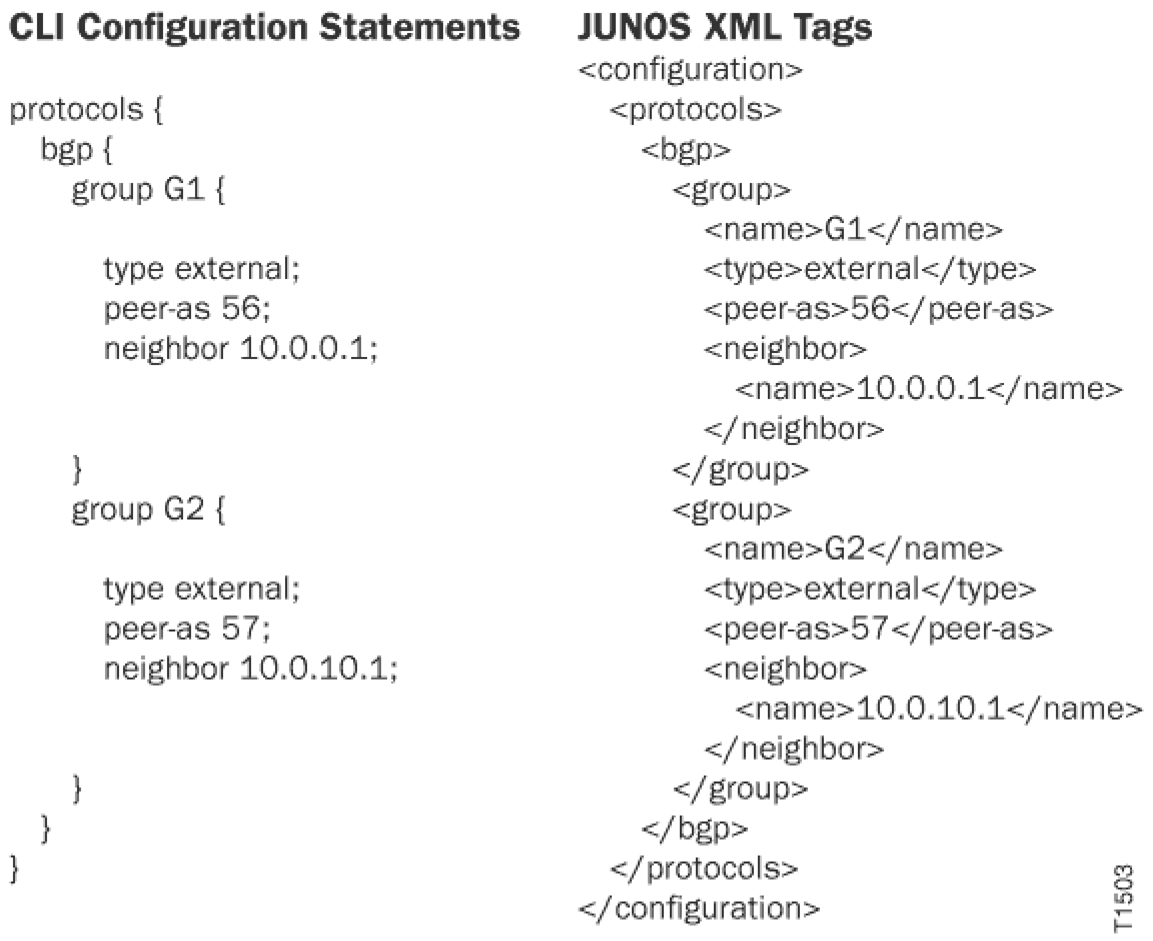
The Junos XML API differs from the CLI in its treatment of the identifier. Because the Junos XML API does not allow container tag elements to contain both other tag elements and untagged character data such as an identifier name, the identifier must be enclosed in a tag element of its own. Most frequently, identifier tag elements for configuration objects are called `<name>`. Some objects have multiple identifiers, which usually have names other than `<name>`. To verify the name of each identifier tag element for a configuration object, consult the entry for the object in the *Junos XML API Configuration Developer Reference*.

NOTE: The Junos OS reserves the prefix `junos-` for the identifiers of configuration groups defined within the `junos-defaults` configuration group. User-defined identifiers cannot start with the string `junos-`.

Identifier tag elements also constitute an exception to the general XML convention that tag elements at the same level of hierarchy can appear in any order; the identifier tag element always occurs first within the container tag element.

The configuration for most objects that have identifiers includes additional leaf statements, which represent other characteristics of the object. For example, each BGP group configured at the `[edit protocols bgp group]` hierarchy level has an associated name (the identifier) and can have leaf statements for other characteristics such as type, peer autonomous system (AS) number, and neighbor address. For information about the Junos XML mapping for leaf statements, see "Mapping for Single-Value and Fixed-Form Leaf Statements" on page 28, "Mapping for Leaf Statements with Multiple Values" on page 29, and "Mapping for Multiple Options on One or More Lines" on page 30.

The following example shows the Junos XML tag elements for configuration statements that define two BGP groups called `<name>` and `<name>`. Notice that the Junos XML `<name>` element that encloses the identifier of each group (and the identifier of the neighbor within a group) does not have a counterpart in the CLI statements.



CLI Configuration Statements

```
protocols {
   bgp {
      group G1 {

         type external;
         peer-as 56;
         neighbor 10.0.0.1;


      }
      group G2 {

         type external;
         peer-as 57;
         neighbor 10.0.10.1;


      }
   }
}
```

JUNOS XML Tags

```
<configuration>
   <protocols>
      <bgp>
         <group>
            <name>G1</name>
            <type>external</type>
            <peer-as>56</peer-as>
            <neighbor>
               <name>10.0.0.1</name>
            </neighbor>
         </group>
         <group>
            <name>G2</name>
            <type>external</type>
            <peer-as>57</peer-as>
            <neighbor>
               <name>10.0.10.1</name>
            </neighbor>
         </group>
      </bgp>
   </protocols>
</configuration>
```

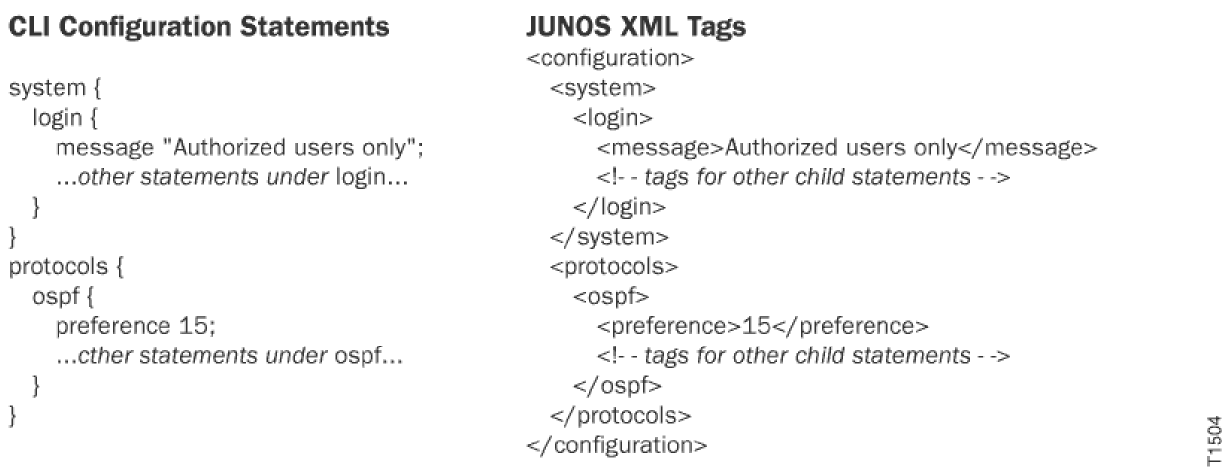
Mapping for Single-Value and Fixed-Form Leaf Statements

A *leaf statement* is a CLI configuration statement that does not contain any other statements. Most leaf statements define a value for one characteristic of a configuration object and have the following form:

```
keyword value;
```

In general, the name of the Junos XML tag element corresponding to a leaf statement is the same as the `keyword` string. The string between the opening and closing Junos XML tags is the same as the *`value`* string.

The following example shows the Junos XML tag elements for two leaf statements that have a keyword and a value: the `message` statement at the `[edit system login]` hierarchy level and the `preference` statement at the `[edit protocols ospf]` hierarchy level.



CLI Configuration Statements

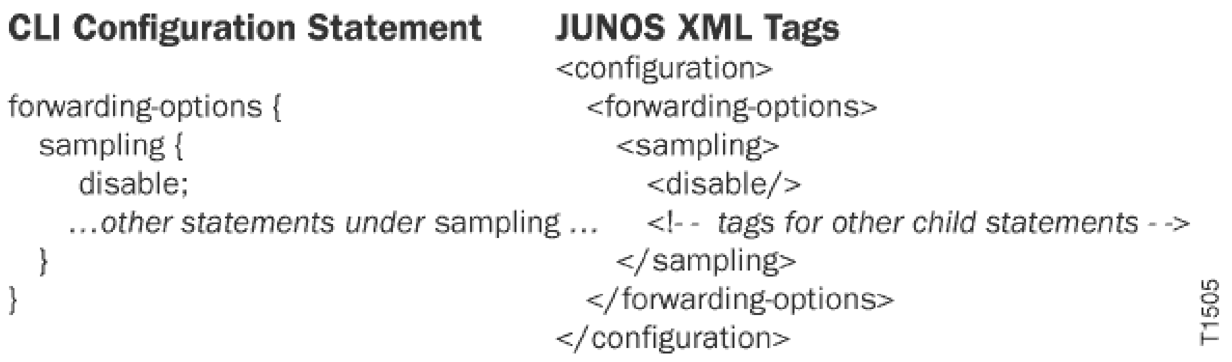
```
system {
  login {
    message "Authorized users only";
    ...other statements under login...
  }
}
protocols {
  ospf {
    preference 15;
    ...cther statements under ospf...
  }
}
```

JUNOS XML Tags

```
<configuration>
  <system>
    <login>
      <message>Authorized users only</message>
      <!- - tags for other child statements - ->
    </login>
  </system>
  <protocols>
    <ospf>
      <preference>15</preference>
      <!- - tags for other child statements - ->
    </ospf>
  </protocols>
</configuration>
```

T1504

Some leaf statements consist of a fixed-form keyword only, without an associated variable-form value. The Junos XML API represents such statements with an empty tag. The following example shows the Junos XML tag elements for the `disable` statement at the `[edit forwarding-options sampling]` hierarchy level.



CLI Configuration Statement

```
forwarding-options {
  sampling {
     disable;
    ...other statements under sampling ...
  }
}
```

JUNOS XML Tags

```
<configuration>
  <forwarding-options>
    <sampling>
      <disable/>
      <!- -  tags for other child statements - ->
    </sampling>
  </forwarding-options>
</configuration>
```

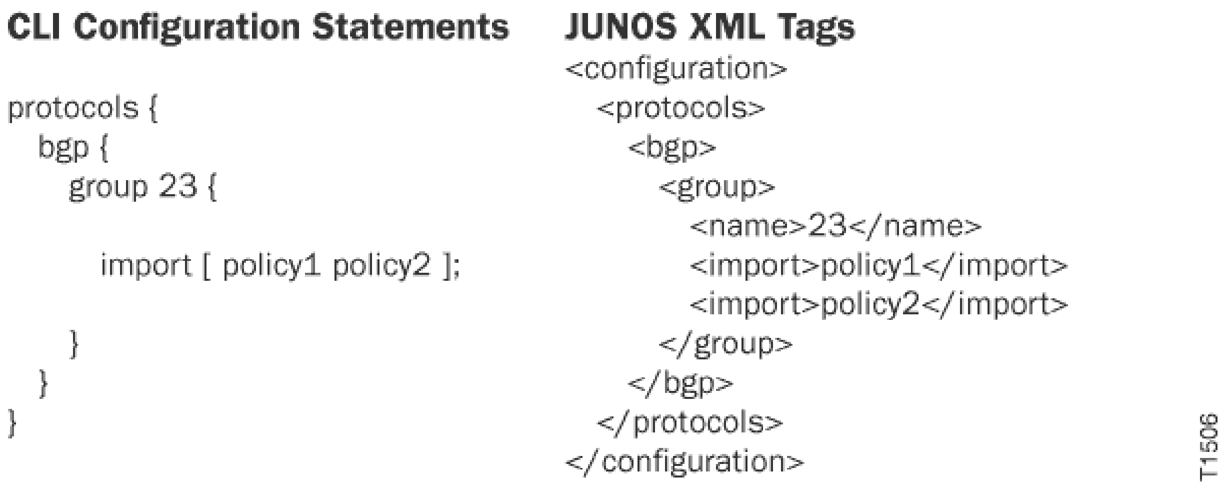
T1505

Mapping for Leaf Statements with Multiple Values

Some Junos OS leaf statements accept multiple values, which can be either user-defined or drawn from a set of predefined values. CLI notation uses square brackets to enclose all values in a single statement, as in the following:

```
statement [ value1 value2 value3 ...];
```

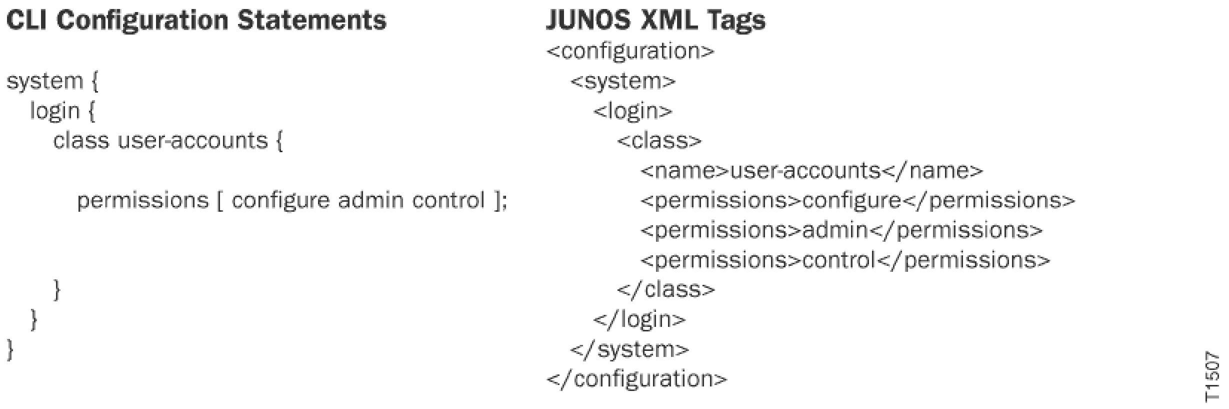
The Junos XML API instead encloses each value in its own tag element. The following example shows the Junos XML tag elements for a CLI statement with multiple user-defined values. The `import` statement imports two routing policies defined elsewhere in the configuration.



```
CLI Configuration Statements       JUNOS XML Tags
                                   <configuration>
protocols {                          <protocols>
  bgp {                                <bgp>
    group 23 {                           <group>
                                           <name>23</name>
      import [ policy1 policy2 ];          <import>policy1</import>
                                           <import>policy2</import>
    }                                    </group>
  }                                    </bgp>
}                                    </protocols>
                                   </configuration>
```

T1506

The following example shows the Junos XML tag elements for a CLI statement with multiple predefined values. The `permissions` statement grants three predefined permissions to members of the `user-accounts` login class.



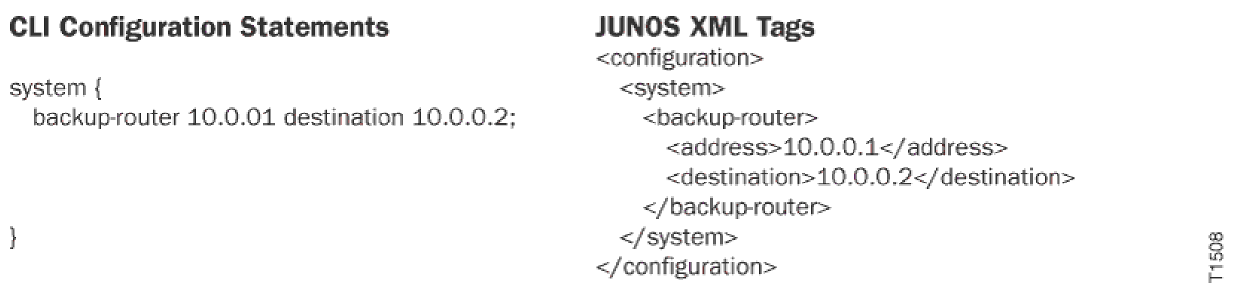
```
CLI Configuration Statements                   JUNOS XML Tags
                                               <configuration>
system {                                         <system>
  login {                                          <login>
    class user-accounts {                            <class>
                                                       <name>user-accounts</name>
      permissions [ configure admin control ];         <permissions>configure</permissions>
                                                       <permissions>admin</permissions>
                                                       <permissions>control</permissions>
    }                                                </class>
  }                                                </login>
}                                                </system>
                                               </configuration>
```

T1507

Mapping for Multiple Options on One or More Lines

For some Junos OS configuration objects, the standard CLI syntax places multiple options on a single line, usually for greater legibility and conciseness. In most such cases, the first option identifies the object and does not have a keyword, but later options are paired keywords and values. The Junos XML API encloses each option in its own tag element. Because the first option has no keyword in the CLI statement, the Junos XML API assigns a name to its tag element.

The following example shows the Junos XML tag elements for a CLI configuration statement with multiple options on a single line. The Junos XML API defines a tag element for both options and assigns a name to the tag element for the first option (10.0.0.1), which has no CLI keyword.



CLI Configuration Statements

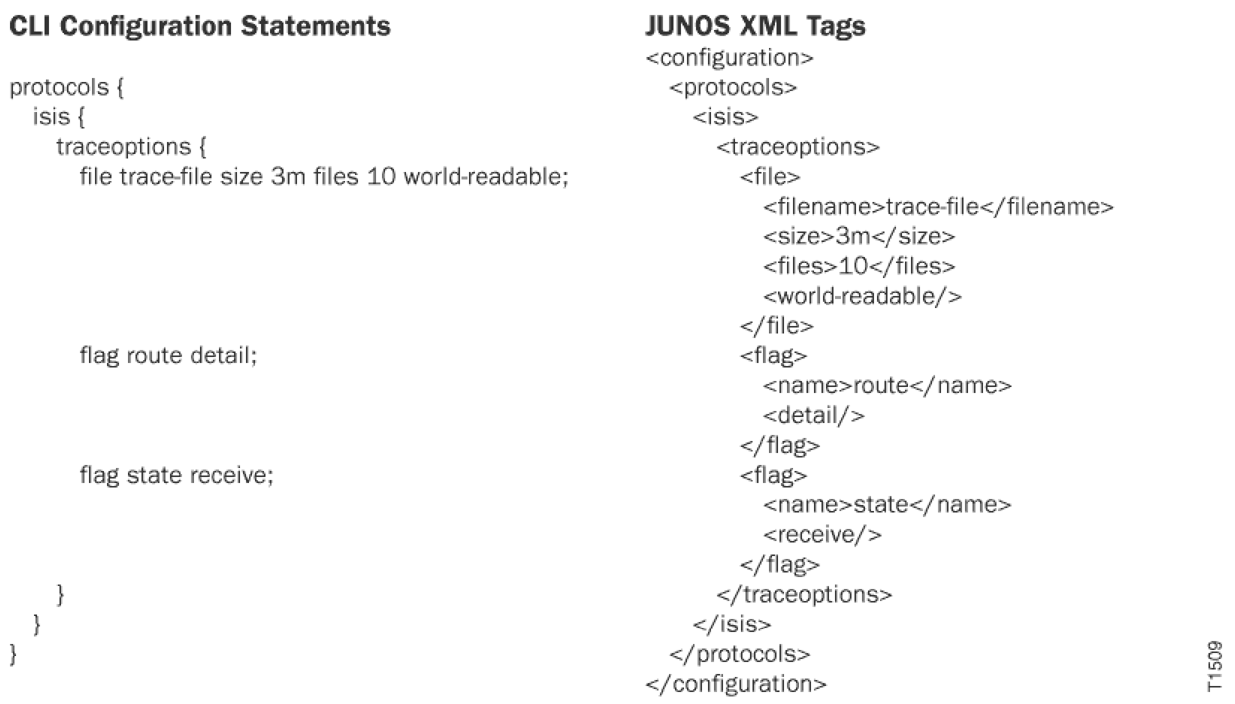
```
system {
  backup-router 10.0.01 destination 10.0.0.2;
}
```

JUNOS XML Tags

```
<configuration>
  <system>
    <backup-router>
      <address>10.0.0.1</address>
      <destination>10.0.0.2</destination>
    </backup-router>
  </system>
</configuration>
```

T1508

The syntax for some configuration objects includes more than one multioption line. Again, the Junos XML API defines a separate tag element for each option. The following example shows Junos XML tag elements for a `traceoptions` statement at the `[edit protocols isis]` hierarchy level. The statement has three child statements, each with multiple options.



CLI Configuration Statements

```
protocols {
  isis {
    traceoptions {
      file trace-file size 3m files 10 world-readable;
      flag route detail;
      flag state receive;
    }
  }
}
```

JUNOS XML Tags

```
<configuration>
  <protocols>
    <isis>
      <traceoptions>
        <file>
          <filename>trace-file</filename>
          <size>3m</size>
          <files>10</files>
          <world-readable/>
        </file>
        <flag>
          <name>route</name>
          <detail/>
        </flag>
        <flag>
          <name>state</name>
          <receive/>
        </flag>
      </traceoptions>
    </isis>
  </protocols>
</configuration>
```

T1509

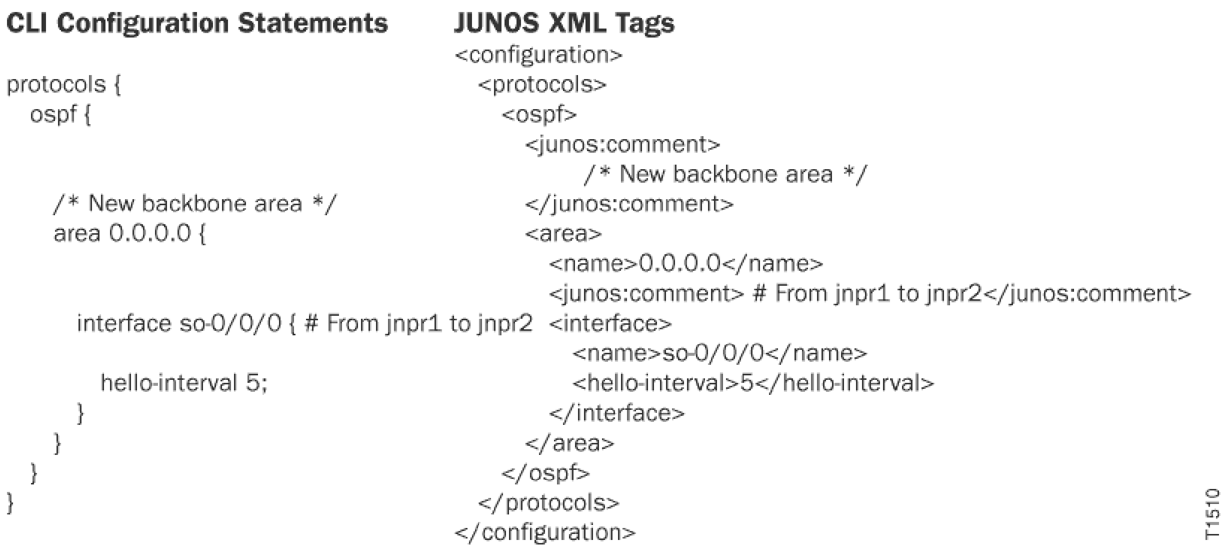
Mapping for Comments About Configuration Statements

A Junos OS configuration can include comments that describe statements in the configuration. In CLI configuration mode, the `annotate` command defines the comment to associate with a statement at the current hierarchy level. You can also use a text editor to insert comments directly into a configuration file. For more information, see the CLI User Guide.

The Junos XML API encloses comments about configuration statements in the `<junos:comment>` element. (These comments are different from the comments that are enclosed in the strings `<!--` and `-->` and are automatically discarded by the protocol server.)

In the Junos XML API, the `<junos:comment>` element immediately precedes the element for the associated configuration statement. (If the tag element for the associated statement is omitted, the comment is not recorded in the configuration database.) The comment text string can include one of the two delimiters that indicate a comment in the configuration database: either the `#` character before the comment or the paired strings `/*` before the comment and `*/` after it. If the client application does not include the delimiter, the protocol server adds the appropriate one when it adds the comment to the configuration. The protocol server also preserves any white space included in the comment.

The following example shows the Junos XML tag elements that associate comments with two statements in a sample configuration statement. The first comment illustrates how including newline characters in the contents of the `<junos:comment>` element (`/* New backbone area */`) results in the comment appearing on its own line in the configuration file. There are no newline characters in the contents of the second `<junos:comment>` element, so in the configuration file the comment directly follows the associated statement on the same line.



```
CLI Configuration Statements          JUNOS XML Tags
                                      <configuration>
protocols {                             <protocols>
  ospf {                                  <ospf>
                                            <junos:comment>
                                                /* New backbone area */
    /* New backbone area */                 </junos:comment>
    area 0.0.0.0 {                          <area>
                                              <name>0.0.0.0</name>
                                              <junos:comment> # From jnpr1 to jnpr2</junos:comment>
      interface so-0/0/0 { # From jnpr1 to jnpr2  <interface>
                                                <name>so-0/0/0</name>
        hello-interval 5;                       <hello-interval>5</hello-interval>
      }                                       </interface>
    }                                       </area>
  }                                       </ospf>
}                                       </protocols>
                                      </configuration>
```

Using NETCONF Configuration Response Tag Elements in NETCONF Requests and Configuration Changes

The NETCONF server encloses its response to each configuration request in `<rpc-reply>` and `<configuration>` tag elements. Enclosing each configuration response within a `<configuration>` tag element contrasts with how the server encloses each different operational response in a tag element named for that type of response—for example, the `<chassis-inventory>` tag element for chassis information or the `<interface-information>` tag element for interface information.

The Junos XML tag elements within the `<configuration>` tag element represent configuration hierarchy levels, configuration objects, and object characteristics, always ordered from higher to deeper levels of the hierarchy. When a client application loads a configuration, it can emit the same tag elements in the same order as the NETCONF server uses when returning configuration information. This consistent representation makes handling configuration information more straightforward. For instance, the client application can request the current configuration, store the NETCONF server's response in a local memory buffer, make changes or apply transformations to the buffered data, and submit the altered configuration as a change to the candidate configuration. Because the altered configuration is based on the NETCONF server's response, it is certain to be syntactically correct.

Similarly, when a client application requests information about a configuration element (hierarchy level or configuration object), it uses the same tag elements that the NETCONF server will return in response. To represent the element, the client application sends a complete stream of tag elements from the top of the configuration hierarchy (represented by the `<configuration>` tag element) down to the requested element. The innermost tag element, which represents the level or object, is either empty or includes the identifier tag element only. The NETCONF server's response includes the same stream of parent tag elements, but the tag element for the requested configuration element contains all the tag elements that represent the element's characteristics or child levels. For more information, see "Request Configuration Data Using NETCONF" on page 400.

The tag streams emitted by the NETCONF server and by a client application can differ in the use of white space, as described in "XML and NETCONF XML Management Protocol Conventions Overview" on page 15.

RELATED DOCUMENTATION

XML and NETCONF XML Management Protocol Conventions Overview | **15**

Map Configuration Statements to Junos XML Tag Elements | **25**

Request Configuration Data Using NETCONF | **400**

2

PART

Manage NETCONF Sessions

CHAPTER 3

NETCONF Session Overview

IN THIS CHAPTER

NETCONF Session Overview

Communication between the NETCONF server and a client application is session based. The server and client explicitly establish a connection and session before exchanging data and close the session and connection when they are finished.

The streams of NETCONF and Junos XML tag elements emitted by the NETCONF server and the client application must each constitute well-formed XML by obeying the structural rules defined in the document type definition (DTD) for the kind of information they are exchanging. The client application must emit tag elements in the required order and only in the allowed contexts.

Client applications can access the NETCONF server by using the SSH protocol and standard SSH authentication mechanisms; by using the TLS protocol, which uses mutual X.509 certificate-based authentication; or by using outbound HTTPS, which uses one-way X.509 certificate based authentication. After authentication, the NETCONF server uses the configured or derived Junos OS username and class to determine whether a client application is authorized to make each request.

The following list outlines the basic structure of a NETCONF session:

1. The client application establishes a connection to the NETCONF server and opens the NETCONF session.

2. The NETCONF server and client application exchange initialization information, which is used to determine if they are using compatible versions of the Junos OS and the NETCONF XML management protocol.

3. The client application sends one or more requests to the NETCONF server and parses its responses.

4. The client application closes the NETCONF session and the connection to the NETCONF server.

For an example of a complete NETCONF session, see "Sample NETCONF Session" on page 136.

RELATED DOCUMENTATION

Generate Well-Formed XML Documents | **37**

Understanding the Client Application's Role in a NETCONF Session

To create a NETCONF session and communicate with the NETCONF server, a client application performs the following procedures, which are described in the indicated sections:

1. Satisfies the prerequisites for the given connection protocol, as described in:
   - "Establish an SSH Connection for a NETCONF Session" on page 41
   - "NETCONF Sessions over Transport Layer Security (TLS)" on page 51
   - "NETCONF and Shell Sessions over Enhanced Outbound HTTPS" on page 68
2. Establishes a connection to the NETCONF server.
   - For NETCONF sessions over SSH, see "Connect to the NETCONF Server Using SSH" on page 117.
   - For NETCONF sessions over TLS, see "How to Establish a NETCONF Session over TLS" on page 56.
   - For NETCONF sessions over outbound HTTPS, see "How to Establish NETCONF and Shell Sessions over Enhanced Outbound HTTPS" on page 71.
3. Opens a NETCONF session, as described in "Start a NETCONF Session" on page 118.
4. Optionally locks the candidate configuration or opens an instance of the ephemeral configuration database.

   Locking the configuration prevents other users or applications from changing it at the same time. For more information, see "Lock and Unlock the Candidate Configuration" on page 132.

   For information about the ephemeral configuration database, see "Understanding the Ephemeral Configuration Database" on page 328 and "Enable and Configure Instances of the Ephemeral Configuration Database" on page 346.

5. Requests operational or configuration information, or changes configuration information, as described in:

    - "Request Operational Information Using NETCONF" on page 386
    - "Request Configuration Data Using NETCONF" on page 400
    - "Edit the Configuration Using NETCONF" on page 277

6. (Optional) Verifies the syntactic correctness of the candidate configuration before attempting to commit it, as described in "Verify the Candidate Configuration Syntax Using NETCONF" on page 322.

7. Commits changes made to the candidate configuration or to an open instance of the ephemeral configuration database, as described in

    - "Commit the Candidate Configuration Using NETCONF" on page 323
    - "Commit the Candidate Configuration Only After Confirmation Using NETCONF" on page 325
    - "Commit and Synchronize Ephemeral Configuration Data Using the NETCONF or Junos XML Protocol" on page 357

8. Unlocks the candidate configuration if it is locked or closes an open instance of the ephemeral configuration database.

    Other users and applications cannot change the candidate configuration while it remains locked. For more information, see "Lock and Unlock the Candidate Configuration" on page 132.

9. Ends the NETCONF session and closes the connection to the device, as described in "End a NETCONF Session and Close the Connection" on page 136.

Generate Well-Formed XML Documents

Each set of NETCONF and Junos XML tag elements emitted by the NETCONF server and a client application within a `<hello>`, `<rpc>`, or `<rpc-reply>` tag element must constitute a well-formed XML document. That is, it must obey the structural rules defined in the document type definition (DTD) for the kind of information being sent. The client application must emit tag elements in the required order and only in the allowed contexts.

NETCONF sessions use a framing mechanism to separate the messages that the NETCONF server and client send within the session. The NETCONF server and client must emit messages using the framing mechanism appropriate for that session. Junos devices support the following framing mechanisms:

- End-of-document character sequence (]]>]]>)—Message separator defined in RFC 4742 *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*

- Chunked framing—Framing mechanism defined in RFC 6242, *Using the NETCONF Protocol over Secure Shell (SSH)*, which encodes all NETCONF messages with chunked framing.

You can configure supported Junos devices to comply with RFC 6242 by configuring the `rfc-compliant` and `version-1.1` statements at the `[edit system services netconf]` hierarchy level. When you enable RFC 6242 compliance and both peers advertise the `:base:1.1` capability, the NETCONF session uses the chunked framing mechanism for the remainder of the session. For more information, see "Configure RFC-Compliant NETCONF Sessions" on page 143.

If the NETCONF session does not use the chunked framing mechanism, the NETCONF server and client applications comply with RFC 4742. In particular, the server and applications send the end-of-document character sequence `]]>]]>` after each XML document. In practice, the client application sends the sequence after the closing `</hello>` tag and each closing `</rpc>` tag, and the NETCONF server sends it after the closing `</hello>` tag and each closing `</rpc-reply>` tag.

The following example demonstrates the general structure of the XML document and the]]>]]> character sequence in a NETCONF session:

```
<!-- generated by the client application -->
<hello | rpc>
    <!-- contents of top-level tag element  -->
</hello | /rpc>
]]>]]>

<!-- generated by the NETCONF server -->
<hello | rpc-reply attributes>
    <!-- contents of top-level tag element  -->
</hello | /rpc-reply>
]]>]]>
```

When the NETCONF session uses chunked framing, the server and client encode all NETCONF messages as chunked following the Augmented Backus-Naur Form (ABNF) rule Chunked-Message. The following example uses chunked framing:

```
<!-- generated by the client application -->
\n#140\n
<?xml version="1.0" encoding="UTF-8"?>\n
<rpc message-id="106"\n
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">\n
   <close-session/>\n
</rpc>
\n##\n
```

```
<!-- generated by the NETCONF server -->
\n#139\n
<?xml version="1.0" encoding="UTF-8"?>\n
<rpc-reply id="106"\n
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">\n
  <ok/>\n
</rpc-reply>
\n##\n
```

Understanding the Request Procedure in a NETCONF Session

You can use the NETCONF XML management protocol and Junos XML API to request information about the status and the current configuration of devices running Junos OS or device running Junos OS Evolved. The tags for operational requests are defined in the Junos XML API and correspond to Junos OS command-line interface (CLI) operational commands. There is a request tag element for many commands in the CLI `show` family of commands.

The tag element for configuration requests is the NETCONF `<get-config>` tag element. It corresponds to the CLI configuration mode `show` command. The Junos XML tag elements that make up the content of both the client application's requests and the NETCONF server's responses correspond to CLI configuration statements, which are described in the Junos OS configuration guides.

In addition to information about the current configuration, client applications can request other configuration-related information, including information about previously committed (rollback) configurations, information about the rescue configuration, or an XML schema representation of the configuration hierarchy.

To request information from the NETCONF server, a client application performs the procedures described in the indicated sections:

1. Establishes a connection to the NETCONF server on the routing, switching, or security platform.

2. Opens a NETCONF session.

3. Optionally locks the candidate configuration or opens an instance of the ephemeral configuration database.

   Locking the configuration prevents other users or applications from changing it at the same time. For more information, see "Lock and Unlock the Candidate Configuration" on page 132.

For information about the ephemeral configuration database, see "Understanding the Ephemeral Configuration Database" on page 328 and "Enable and Configure Instances of the Ephemeral Configuration Database" on page 346.

4. Makes any number of requests one at a time, freely intermingling operational and configuration requests. See "Request Operational Information Using NETCONF" on page 386 and "Request Configuration Data Using NETCONF" on page 400. The application can also intermix requests with configuration changes.

5. Accepts the tag stream emitted by the NETCONF server in response to each request and extracts its content, as described in "Parse the NETCONF Server Response" on page 127.

6. Unlocks the candidate configuration, if it is locked, or closes an open instance of the ephemeral configuration database.

   Other users and applications cannot change the candidate configuration while it remains locked. For more information, see "Lock and Unlock the Candidate Configuration" on page 132

7. Ends the NETCONF session and closes the connection to the device, as described in "End a NETCONF Session and Close the Connection" on page 136.

CHAPTER 4

Manage NETCONF Sessions

IN THIS CHAPTER

Establish an SSH Connection for a NETCONF Session

IN THIS SECTION

Understanding NETCONF Sessions over SSH

You can use the SSH protocol to establish connections between a *configuration management server* (CMS) and a Junos device. You use a configuration management server to manage the Junos device remotely.

You can use the following options to establish an SSH connection between the configuration management server and the Junos device.

- SSH—The configuration management server initiates an SSH session with the Junos device.
- Outbound SSH—The Junos device initiates, establishes, and maintains an SSH connection with a predefined set of configuration management servers. Use this option when the configuration management server cannot initiate an SSH connection because of network restrictions (such as a firewall).

NOTE: Junos OS includes a customized implementation of OpenSSH for device management. Security fixes are backported as needed, independent of the OpenSSH version numbers. The version displayed in CLI output (for example, `show version`) might not reflect all applied patches. Always refer to Juniper Security Advisories (JSAs) for vulnerability impact assessments.

Common Prerequisites for NETCONF Sessions over SSH or Outbound SSH

IN THIS SECTION

For NETCONF sessions that use either SSH or outbound SSH, you must install SSH software on the configuration management server and enable the NETCONF service on the Junos device. See the following sections for detailed instructions:

Install SSH Software on the Configuration Management Server

For SSH connections, the configuration management server (CMS) handles the SSH connection with the Junos device. For outbound SSH connections, the Junos device establishes the SSH connection to the configuration management server, and then the configuration management server takes control of the SSH session.

To establish an SSH or outbound SSH connection with a device, the configuration management server must have SSH software installed locally. For information about obtaining and installing SSH software, see:

- http://www.ssh.com
- http://www.openssh.com

Enable NETCONF Service over SSH

To establish NETCONF sessions on a Junos device, you must enable the NETCONF service. You can configure the NETCONF server to accept NETCONF sessions on the following ports:

- Default NETCONF port (830) or a user-defined port
- Default SSH port (22)

We recommend that you use the default NETCONF port because the device can identify and filter NETCONF traffic more effectively. Alternatively, you can configure the device to accept NETCONF sessions on a port number of your choosing instead of the default NETCONF port. The defined port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests.

If you enable NETCONF and you also enable SSH services on the device, the device accepts NETCONF sessions on both the default SSH port and the configured NETCONF port (default or user-defined port). For added security, you can configure event policies that utilize `UI_LOGIN_EVENT` information to effectively disable the SSH port from accepting NETCONF sessions or to further restrict NETCONF server access on a port.

To enable NETCONF service over SSH on a Junos device:

1. Enable the NETCONF service on either the default NETCONF port (830) or a user-defined port:

    - To use the default NETCONF port (830), include the `netconf ssh` statement at the `[edit system services]` hierarchy level:

    ```
    [edit system services]
    user@host# set netconf ssh
    ```

    - To use a specific port, configure the `port` statement with the port number at the `[edit system services netconf ssh]` hierarchy level.

    ```
    [edit system services]
    user@host# set netconf ssh port port-number
    ```

The *port-number* can range from 1 through 65535. The configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests.

NOTE: Although you can configure NETCONF on any port from 1 through 65535, you should not configure access on a port that is normally assigned for another service. This practice avoids potential resource conflicts. If you configure a port assigned for another service, such as FTP, and that service is enabled, a `commit check` does not reveal a resource conflict or issue any warning message.

2. (Optional) To also enable access to the NETCONF SSH subsystem using the default SSH port (22), include the `ssh` statement at the `[edit system services]` hierarchy level.

```
[edit system services]
user@host# set ssh
```

This configuration enables SSH access to the device for all users and applications.

NOTE: In certain releases, the default behavior is to restrict the root user from using the SSH service. In those releases, you must configure the `root-login allow` statement at the `[edit system services ssh]` hierarchy level to enable the root user to open NETCONF sessions over SSH.

3. (Optional) Configure the device to disconnect unresponsive NETCONF clients.

Specify the timeout interval (in seconds) after which, if no data has been received from the client, the sshd process requests a response. Additionally, specify the threshold of missed client-alive responses that triggers a disconnect.

```
[edit system services]
user@host# set netconf ssh client-alive-interval 10
user@host# set netconf ssh client-alive-count-max 10
```

NOTE: Statements configured at the `[edit system services netconf ssh]` hierarchy level apply only to NETCONF sessions that connect through the default port (830) or through the user-defined port that is configured at the same hierarchy level.

4. Commit the configuration:

```
[edit]
user@host# commit
```

5. Repeat the preceding steps on each Junos device where the client application establishes NETCONF sessions.

Prerequisites for Establishing an SSH Connection for NETCONF Sessions

IN THIS SECTION

Before the configuration management server can establish an SSH connection with a Junos device, you must satisfy the common requirements discussed in:

- "Common Prerequisites for NETCONF Sessions over SSH or Outbound SSH" on page 42

You must also satisfy the requirements discussed in the following sections:

Configure a User Account for the Client Application on Junos Devices

The configuration management server must log in to the Junos device to establish a NETCONF session. Thus, the configuration management server needs a user account on each device where it establishes a NETCONF session. The following instructions explain how to create a local user account on Junos devices. Alternatively, you can skip this section and enable authentication through RADIUS or TACACS +.

To create a local user account:

1. Configure the `user` statement and specify a username. Include the `class` statement, and specify a login class that has the permissions required for all actions to be performed by the application.

```
[edit system login]
user@host# set user username class class-name
```

2. Optionally, include the `full-name` and `uid` statements at the `[edit system login user *username*]` hierarchy level.
3. Commit the configuration to activate the user account on the device.

```
[edit]
user@host# commit
```

4. Repeat the preceding steps on each Junos device where the client application establishes NETCONF sessions.

Configure a Public/Private Keypair or Password for the Junos OS User Account

The configuration management server needs an SSH public/private keypair, a text-based password, or both to authenticate with the NETCONF server. A keypair is sufficient if the account is used only to connect to the NETCONF server through SSH. If the account is also used to access the device in other ways (for login on the console, for example), it must have a text-based password. The password is also used (the SSH server prompts for it) if key-based authentication is configured but fails.

NOTE: You can skip this section if you have chosen to enable authentication through RADIUS or TACACS+.

To create a text-based password:

1. Include either the `plain-text-password` or `encrypted-password` statement at the `[edit system login user *username* authentication]` hierarchy level.

   To enter a password as text, issue the following command. You are prompted for the password, which the device encrypts before storing.

```
[edit system login user username authentication]
user@host# set plain-text-password
New password: password
Retype new password: password
```

   To enter a password that you previously created and hashed using MD5 or SHA-1, issue the following command:

```
[edit system login user username authentication]
user@host# set encrypted-password "password"
```

2. Commit the configuration.

   ```
   [edit system login user username authentication]
   user@host# commit
   ```

3. Repeat the preceding steps on each device where the client application establishes NETCONF sessions.

To create an SSH public/private keypair, perform the following steps:

1. On the configuration management server where the client application runs, issue the `ssh-keygen` command in the standard command shell and provide the appropriate arguments.

   ```
   user@cms:~$ ssh-keygen options
   ```

   For example:

   ```
   netconf-user@cms:~$ ssh-keygen -t rsa -b 4096
   Generating public/private rsa key pair.
   Enter file in which to save the key (/home/netconf-user/.ssh/id_rsa):
   Created directory '/home/netconf-user/.ssh'.
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in /home/netconf-user/.ssh/id_rsa
   Your public key has been saved in /home/netconf-user/.ssh/id_rsa.pub
   ...
   ```

   For more information about `ssh-keygen` options, see the man page for the `ssh-keygen` command.

2. Associate the public key with the Junos OS login account.

   ```
   [edit system login user username authentication]
   user@host# set load-key-file URL
   ```

   Junos OS copies the contents of the specified file onto the device. *URL* is the path to the file that contains one or more public keys. The `ssh-keygen` command by default stores each public key in a file in the **.ssh** subdirectory of the user home directory; the filename depends on the encoding and SSH version. For information about specifying URLs, see the CLI User Guide.

NOTE: Alternatively, you can include the `ssh-rsa` statement at the `[edit system login user` *`account-name`* `authentication]` hierarchy level. We recommend using the `load-key-file` statement, however, because it eliminates the need to type or cut-and-paste the public key on the command line.

3. Commit the configuration.

```
[edit]
user@host# commit
```

4. Repeat Step "2" on page 47 and Step "3" on page 48 on each Junos device where the client application establishes NETCONF sessions.

Access the Keys or Password with the Client Application

The client application must be able to access the configured keypair or password and provide it when the NETCONF server prompts for it.

There are several methods for enabling the application to access the key or password:

- If public/private keys are used, the ssh-agent program runs on the device where the client application runs, and handles the private key.
- When a user starts the application, the application prompts the user for the password and stores it temporarily in a secure manner.
- The password is stored in encrypted form in a secure local-disk location or in a secured database.

Prerequisites for Establishing an Outbound SSH Connection for NETCONF Sessions

IN THIS SECTION

To enable a configuration management server to establish an outbound SSH connection to the NETCONF server, you must satisfy the common requirements discussed in:

- "Common Prerequisites for NETCONF Sessions over SSH or Outbound SSH" on page 42

You must also satisfy the requirements discussed in the following sections:

Configure the Junos Device for Outbound SSH

To configure the Junos device for outbound SSH:

1. At the `[edit system services ssh]` hierarchy level, set the SSH `protocol-version` to v2:

   ```
   [edit system services ssh]
   user@host# set protocol-version v2
   ```

2. Generate or obtain a public/private keypair for the Junos device. This keypair is used to encrypt the data transferred across the SSH connection.
3. If you are manually installing the public key on the configuration management server, transfer the public key to the CMS.
4. At the `[edit system services]` hierarchy level, include the `outbound-ssh` configuration hierarchy and any required statements.

   ```
   [edit system services]
   outbound-ssh {
       client client-id {
           address {
               port port-number;
               retry number;
               timeout seconds;
           }
           device-id device-id;
           keep-alive {
               retry number;
               timeout seconds;
           }
           reconnect-strategy (in-order | sticky);
           secret password;
           services  netconf;
       }
   }
   ```

   For detailed information about each of the options, see *outbound-ssh*.

5. Commit the configuration:

```
[edit]
user@host# commit
```

Receive and Manage the Outbound SSH Initiation Sequence on the Client

When you configure a Junos device for outbound SSH, the device attempts to maintain a constant connection with a configuration management server. Whenever an outbound SSH session is not established, the device sends an outbound SSH initiation sequence to a configuration management server listed in the device's configuration management server list. Before establishing a connection with the device, each configuration management server must be set up to receive this initiation sequence, establish a TCP connection with the device, and transmit the device identity back to the device.

The initiation sequence takes one of two forms, depending on how you chose to handle the Junos OS server's public key.

If the public key is installed manually on the configuration management server, the initiation sequence takes the following form:

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: <device-id>\r\n
```

If the device forwards the public key to the configuration management server during the initialization sequence, the sequence takes the following form:

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: : <device-id>\r\n
HOST-KEY: <pub-host-key>\r\n
HMAC: <HMAC(pub-SSH-host-key,<secret>)>\r\n
```

RELATED DOCUMENTATION

Remote Access Overview

Junos OS User Accounts

NETCONF Sessions over Transport Layer Security (TLS)

SUMMARY

Network Configuration Protocol (NETCONF) clients can use the Transport Layer Security (TLS) protocol with mutual X.509 certificate-based authentication to establish a NETCONF session with supported Junos devices.

IN THIS SECTION

Understanding NETCONF-over-TLS Connections

IN THIS SECTION

Benefits of NETCONF over TLS

- Enables remote management of devices using mutual certificate-based authentication
- Enables you to more easily manage networks on a larger scale than when using NETCONF over SSH
- Uses public-key infrastructure to provide mutual TLS certificate-based authentication for both the client and the server
- Secures the connection and exchange of NETCONF messages
- Ensures data integrity for exchanged messages

NETCONF over TLS Overview

You can establish a Network Configuration Protocol (NETCONF) session over Transport Layer Security (TLS) on certain Junos devices, as an alternative to establishing a NETCONF session over SSH. TLS is a cryptographic protocol that uses mutual certificate-based authentication and provides a secure and reliable connection between two devices. It is a successor to the Secure Sockets Layer (SSL) protocol.

When you establish a NETCONF session over TLS, the NETCONF server acts as the TLS server, and the NETCONF client is the TLS client.

NETCONF sessions over TLS provide some advantages over sessions that use SSH. Whereas SSH authenticates a client by using credentials (username and password) or keys, TLS uses certificates to mutually authenticate both the client and the server. Certificates can provide additional information about a client, and they can be used to securely authenticate one device to another. Thus, while NETCONF sessions over SSH work well for manually managing individual devices, NETCONF sessions that use TLS enable secure device-to-device communication to better manage and automate devices in large-scale networks.

NETCONF-over-TLS sessions with Junos devices have the following requirements:

- NETCONF client that supports TLS version 1.2
- The server and client must have X.509 public key certificates that are signed by a certificate authority
- The Junos public key infrastructure (PKI) must have the appropriate local and CA certificates loaded
- The Junos device is configured for NETCONF over TLS and defines a default or specific certificate-to-NETCONF-username mapping for a client
- The NETCONF username corresponds to a valid Junos OS user account

TLS uses X.509 digital certificates for server and client authentication. A digital certificate is an electronic means for verifying your identity through a trusted third party, known as a *certificate authority* or *certification authority (CA)*. A certificate authority issues digital certificates, which can be used to establish a secure connection between two endpoints through certificate validation. The X.509 standard defines the format for the certificates. To establish a NETCONF session over TLS on supported Junos devices, both the server and the client must have a valid X.509 certificate, and the certificates must be signed by a CA. Self-signed certificates cannot be used to establish NETCONF sessions over TLS.

The Junos OS PKI provides an infrastructure for digital certificate management. To establish a TLS connection, you must install the following in the Junos OS PKI:

- NETCONF server's local certificate and its intermediate CA certificates

  NOTE: If the server certificate chain does not include intermediate CAs, you must configure the root CA certificate.

- NETCONF client's root CA certificate required to validate the NETCONF client certificate or certificate chain

After the server verifies the identity of the client and establishes the TLS connection, it must derive the NETCONF username for that client before it can establish the NETCONF session. The NETCONF username is the Junos user account under whose access privileges and permissions the NETCONF operations are performed. You can configure a list of client certificate-to-NETCONF username mappings, and you can also configure a default NETCONF username mapping. Junos OS uses the default mapping when a client certificate does not match any of the configured clients. If the server extracts a valid NETCONF username, it then establishes the NETCONF session. For more information about deriving the NETCONF username, see "Understanding the TLS Client to NETCONF Username Mapping" on page 53.

The Junos process tls-proxyd handles the TLS connection. It performs the TLS handshake, encrypts and decrypts the traffic, determines the NETCONF username, and fetches the authorization parameters for the NETCONF user. The tls-proxyd process works in conjunction with the management process (mgd) to create and manage the NETCONF session. The NETCONF-over-TLS session workflow is outlined in "NETCONF-over-TLS Connection Workflow" on page 55.

For more information about NETCONF over TLS, see RFC 7589, *Using the NETCONF Protocol over Transport Layer Security (TLS) with Mutual X.509 Authentication*.

For more information about the Transport Layer Security protocol, see RFC 5246, *The Transport Layer Security (TLS) Protocol Version 1.2*.

Understanding the TLS Client to NETCONF Username Mapping

The authenticated identity of the NETCONF-over-TLS client is the NETCONF username. Junos devices execute the NETCONF operations under the account privileges of this user. You can configure the method used to derive the NETCONF username for individual clients, and you can also define a default method to derive the NETCONF username for those clients that do not match a configured client.

You can configure the mapping of client certificates to NETCONF usernames at the `[edit system services netconf tls client-identity]` hierarchy level. For each client, you configure the certificate fingerprint and a map type. If the fingerprint of a client certificate matches a configured fingerprint, Junos OS uses the corresponding map type to derive the NETCONF username. You can configure only one fingerprint per client, and each client fingerprint must be unique. For example:

```
netconf {
    tls {
        client-identity client1 {
            fingerprint
04:D2:96:AF:89:AB:33:A4:F9:5C:0F:34:9E:FC:67:2D:98:C6:08:9B:E8:6C:DE:63:60:1C:F6:CD:1A:43:5A:30:A
D;
            map-type specified;
            username netconf-user;
        }
        client-identity client2 {
            fingerprint
04:95:71:45:4F:56:10:CA:B1:89:A3:8C:5D:89:CC:BD:01:37:03:EC:B5:4A:55:22:AD:49:DA:9B:D8:8B:3A:21:1
2;
            map-type san-dirname-cn;
        }
    }
}
```

The configured certificate fingerprint uses x509c2n:tls-fingerprint format as defined in RFC 7407, *A YANG Data Model for SNMP Configuration*. In this format, the first octet is the hashing algorithm identifier, and the remaining octets are the result of the hashing algorithm. The hashing algorithm identifier, which is shown here for reference, is defined in RFC 5246, *The Transport Layer Security (TLS) Protocol Version 1.2*.

- md5: 1
- sha1: 2
- sha224: 3
- sha256: 4
- sha384: 5
- sha512: 6

You can also configure a default mapping for the NETCONF username at the `[edit system services netconf tls default-client-identity]` hierarchy level. If the fingerprint of a client certificate does not match any configured clients, the Junos device uses the default map type to derive the NETCONF username.

Junos devices support the following map types:

- `san-dirname-cn`—Use the common name (CN) defined for the SubjectAltName's (SAN) DirName field (`DirName:/CN`) in the client certificate as the NETCONF username.
- `specified`—Use the NETCONF username defined in the `username` statement at the same hierarchy level.

After the server verifies the identity of the client and establishes the TLS connection, it derives the NETCONF username. It first matches the fingerprint for each configured client against the fingerprint of the presented certificate. If there is a match, it uses the corresponding map type to derive the NETCONF username. If none of the configured fingerprints match that of the client's certificate, the default map type is used to derive the NETCONF username.

After the server determines the username, it fetches the authorization for the user locally or remotely. The username must either have a user account defined locally on the device, or it must be authenticated by a Lightweight Directory Access Protocol (LDAP) server, which then maps it to a user template account that is defined locally on the Junos device. If the extracted username is not a valid local or remote user, then the TLS connection is terminated.

NETCONF-over-TLS Connection Workflow

The Junos device acts as the TLS and NETCONF server. The server listens for incoming NETCONF-over-TLS connections on TCP port 6513. The NETCONF client, which is also the TLS client, initiates a connection with the server on that port.

The client and server perform the following actions to establish and use the NETCONF session over TLS:

1. The client sends a TLS ClientHello message to initiate the TLS handshake.
2. The server sends a ServerHello message, the server certificate chain, and a CertificateRequest message to request a certificate from the client.
3. The client verifies the identity of the server and sends the client certificate chain.
4. The server verifies the client certificate chain with the client's root CA, which has been preconfigured on the server.
5. The server derives the NETCONF username for that client.
6. If the NETCONF username is valid, the server starts the NETCONF session, and the server and client exchange NETCONF `<hello>` messages.
7. The client performs NETCONF operations using the access privileges and permissions of the NETCONF user.
8. The client executes the `<close-session>` operation to end the NETCONF session, which subsequently closes the TLS connection.

The server fails to establish the NETCONF session over TLS in the following scenarios:

- The server or client certificate is expired or self-signed.
- The client doesn't provide a certificate.
- The client doesn't send its intermediate CA certificates.
- The client's root CA certificate is not configured on the server.
- The server cannot map the client certificate to a configured or default map type to derive the NETCONF username.
- The server uses the `san-dirname-cn` map type to derive the NETCONF username for the client, but the client's certificate does not specify a username in the corresponding field.

SEE ALSO

PKI Components In Junos OS

How to Establish a NETCONF Session over TLS

IN THIS SECTION

A *network management system* (NMS) is used to remotely manage the Junos device. You can establish a NETCONF session over TLS between a network management system and supported Junos devices. The NMS is the NETCONF and TLS client, and the Junos device is the NETCONF and TLS server.

Before the client and server can establish a NETCONF session over TLS, you must satisfy the requirements discussed in the following sections:

Install TLS Client Software on the Network Management System

To establish a NETCONF session using TLS, the network management system must first establish a TLS connection with the Junos device. Thus, the network management system requires software for managing the TLS protocol. For example, you can install and use the OpenSSL toolkit, which is a toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is licensed under an Apache-style license.

For more information about OpenSSL, see https://www.openssl.org.

Obtain X.509 Certificates for the Server and Client

The TLS protocol uses X.509 public key certificates to authenticate the identity of the server and the client. To establish a NETCONF session over TLS, both the server and the client must have an X.509

certificate, and the certificate must be signed by a valid certificate authority (CA). Self-signed certificates are not accepted for NETCONF sessions over TLS.

To use OpenSSL to obtain a certificate for the NETCONF client:

1. Generate a private key, and specify the key length in bits.

```
user@nms:~$ openssl genrsa -out client.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
......................................................+++++
............+++++
e is 65537 (0x010001)
```

NOTE: Junos devices do not support using Elliptic Curve Digital Signature Algorithm (ECDSA) keys in NETCONF sessions over TLS.

2. If you are defining the NETCONF username in the client's certificate, update your **openssl.cnf** or equivalent configuration file to define the `subjectAltName=dirName` extension and specify the NETCONF username.

```
user@nms:~$ cat openssl.cnf
# OpenSSL configuration file.
...
[usr_cert]
subjectAltName=dirName:dir_sect

[dir_sect]
CN=netconf-user
...
```

3. Generate a certificate signing request (CSR), which contains the entity's public key and information about their identity.

```
user@nms:~$ openssl req -new -key client.key -out client.csr -sha256
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
```

```
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Sunnyvale
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Juniper
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:netconf-tls-client.example.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

4. Generate the certificate by doing one of the following:

   - Submit the CSR to a certificate authority to request an X.509 certificate, and provide the configuration file to include any additional extensions.

   - Sign the CSR with a CA to generate the client certificate, and include the `-extfile` and `-extensions` options if you need to reference your configuration file and extensions.

     ```
     user@nms:~$ openssl x509 -req -in client.csr -CA clientRootCA.crt -CAkey clientRootCA.key -
     CAcreateserial -out client.crt -days 365 -extensions usr_cert -extfile openssl.cnf
     Signature ok
     subject=C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = netconf-tls-client.example.com
     Getting CA Private Key
     ```

5. Verify that the Common Name (CN) field and extensions, if provided, are correct.

```
user@nms:~$ openssl x509 -text -noout -in client.crt
Certificate:
    Data:
        Version: 3 (0x2)
        ...
        Subject: C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = netconf-tls-
client.example.com
        ...
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DirName:/CN=netconf-user
    ...
```

Similarly, generate the server certificate.

1. Generate a private key, and specify the key length in bits.

```
user@nms:~$ openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................................................+++++
.............+++++
e is 65537 (0x010001)
```

2. Generate a certificate signing request (CSR).

```
user@nms:~$ openssl req -new -key server.key -out server.csr -sha256 -subj "/C=US/ST=CA/
L=Sunnyvale/O=Juniper/CN=host.example.com"
```

3. Generate the certificate by doing one of the following:

   - Submit the CSR to a certificate authority to request an X.509 certificate.
   - Sign the CSR with a CA to generate the server certificate.

```
user@nms:~$ openssl x509 -req -in server.csr -CA serverIntCA.crt -CAkey serverIntCA.key -
CAcreateserial -out server.crt -days 365
Signature ok
subject=C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = host.example.com
Getting CA Private Key
```

The Junos OS public key infrastructure (PKI) provides an infrastructure for digital certificate management. You can also use the Junos OS PKI to generate the required key pair and CSR for the server's local certificate. For information about the Junos OS PKI and the different methods for obtaining certificates, see Digital Certificates with PKI Overview and related documentation.

Install the Server's Local Certificate in the Junos PKI

The server's local certificate is the X.509 certificate for the Junos device that is acting as the NETCONF and TLS server. You must install the local certificate for the device in the Junos PKI.

To install the server's local certificate in the Junos PKI:

1. Copy the certificate and private key to the Junos device.
2. Load the certificate from the specified file using the Junos PKI.

Define a unique certificate identifier, and specify the file paths to the certificate and the private key or key pair. For example:

```
user@host> request security pki local-certificate load certificate-id netconf-server-cert
filename /var/tmp/server.crt key /var/tmp/server.key
Local certificate loaded successfully
```

3. (Optional) Verify the certificate.

```
user@host> show security pki local-certificate certificate-id netconf-server-cert
Certificate identifier: netconf-server-cert
  Issued to: host, Issued by: C = US, ST = California, L = Sunnyvale, O = ServerIntCA, CN =
ServerIntCA
  Validity:
    Not before: 03- 6-2020 22:32 UTC
    Not after: 03- 6-2021 22:32 UTC
  Public key algorithm: rsaEncryption(2048 bits)
  Keypair Location: Keypair generated locally
```

Install the CA Certificates in the Junos PKI

A digital certificate is an electronic means for verifying your identity through a trusted third party, known as a *certificate authority (CA)*. When establishing a NETCONF session over TLS, the client and server must each have an X.509 digital certificate to authenticate their identity. You must configure the root CA certificate required to validate the client certificate in the Junos public key infrastructure (PKI). You must also configure any CAs required to validate the server's local certificate in the Junos PKI. Thus, for each CA, you configure a certificate authority profile and load the corresponding CA certificate and certificate revocation list (CRL). This configuration enables the Junos device to validate a certificate against the CA.

NOTE: If the server certificate chain does not include intermediate CAs, you must configure the root CA certificate. Otherwise, you only need to configure the intermediate CAs.

To manually configure a CA profile and load the corresponding CA certificate and CRL:

1. Download the CA certificates and any required CA certificate revocation lists (CRLs) to the Junos device.

2. Configure a trusted CA profile for each required CA, for example:

```
[edit security pki]
user@host# set ca-profile clientRootCA ca-identity clientRootCA
user@host# set ca-profile serverRootCA ca-identity serverRootCA
user@host# set ca-profile serverIntCA ca-identity serverIntCA
user@host# commit and-quit
```

3. Load the CA certificate associated with the client’s root CA profile in the Junos PKI, and specify the location of the certificate file.

```
user@host> request security pki ca-certificate load ca-profile clientRootCA filename /var/tmp/
clientRootCA.crt
Fingerprint:
  93:cc:d4:bb:ce:6b:e5:8d:91:e2:f9:46:7c:f8:a5:52:87:88:b5:28 (sha1)
  03:18:f4:42:38:fd:ad:c4:73:78:06:cd:45:2a:de:e2 (md5)
Do you want to load this CA certificate ? [yes,no] (no) yes

CA certificate for profile clientRootCA loaded successfully
```

4. Load the CA certificates associated with the server’s CA profile in the Junos PKI, and specify the location of the certificate file.

   - If the certificate chain only has a root CA, load the root CA certificate.

```
user@host> request security pki ca-certificate load ca-profile serverRootCA
filename /var/tmp/serverRootCA.crt
Fingerprint:
  af:67:c6:f0:7c:2d:11:35:72:0e:c3:b3:76:ee:63:57:d4:81:a4:77 (sha1)
  2a:87:1f:f8:9d:67:4c:d3:94:d2:b1:29:14:e0:90:2e (md5)
Do you want to load this CA certificate ? [yes,no] (no) yes

CA certificate for profile serverRootCA loaded successfully
```

   - If the certificate chain includes intermediate CAs, you only need to load the intermediate CA certificates.

```
user@host> request security pki ca-certificate load ca-profile serverIntCA
filename /var/tmp/serverIntCA.crt
Fingerprint:
  7c:a2:59:0e:6d:8b:6a:c5:da:e2:73:73:b0:cc:4a:28:39:dd:a2:52 (sha1)
  57:03:85:ef:eb:e8:72:a6:70:a0:c3:c9:35:e8:6a:eb (md5)
Do you want to load this CA certificate ? [yes,no] (no) yes

CA certificate for profile serverIntCA loaded successfully
```

5. Load the CRL for a given CA profile where required, for example:

```
user@host> request security pki crl load ca-profile clientRootCA filename /var/tmp/revoke.crl
```

NOTE: If you do not configure a certificate revocation list for a given CA profile, then you must disable revocation checks by configuring the `revocation-check disable` statement at the `[edit security pki ca-profile *profile-name*] hierarchy level`.

6. (Optional) Verify the CA certificate.

```
user@host> show security pki ca-certificate ca-profile clientRootCA detail
LSYS: root-logical-system
  CA profile: clientRootCA
Certificate identifier: clientRootCA
  Certificate version: 3
...
```

Enable the NETCONF Service over TLS

To enable NETCONF over TLS:

1. Configure the server's local certificate ID, and reference the ID that was defined when the certificate was installed.

```
[edit system services netconf tls]
user@host# set local-certificate netconf-server-cert
```

2. Define how the server should derive the NETCONF username for a given client.

   - You can define the mapping for an individual client, as described in "Configure the TLS Client-to-NETCONF Username Mapping" on page 63.

   - You can also define a default mapping that is used when a client does not match any of the configured clients. See "Configure the Default NETCONF Username Mapping" on page 64.

3. (Optional) Configure trace options for NETCONF sessions over TLS, for example:

```
[edit system services netconf tls]
user@host# set traceoptions file size 10m
user@host# set traceoptions file files 2
user@host# set traceoptions flag all
```

4. Commit the configuration.

```
[edit system services netconf tls]
user@host# commit
```

Configure the TLS Client-to-NETCONF Username Mapping

You can define the mapping between the client certificate and the NETCONF username for specific clients. If you do not define a mapping for a specific client, then you must define a default mapping in order for the client to establish a NETCONF session over TLS.

To define the mapping to derive the NETCONF username for a given client:

1. Determine the fingerprint for the client's certificate by executing the command appropriate for your environment on the network management system and the format of the certificate, for example:

```
user@nms:~$ openssl x509 -noout -fingerprint -sha256 -in client.crt
SHA256
Fingerprint=D2:96:AF:89:AB:33:A4:F9:5C:0F:34:9E:FC:67:2D:98:C6:08:9B:E8:6C:DE:63:60:1C:F6:CD:1
A:43:5A:30:AD
```

2. Determine the fingerprint's hashing algorithm identifier as defined in RFC 5246, *The Transport Layer Security (TLS) Protocol Version 1.2*.

   This examples uses the SHA-256 hashing algorithm, which corresponds to the identifier value of 4.

   - md5: 1
   - sha1: 2
   - sha224: 3
   - sha256: 4
   - sha384: 5
   - sha512: 6

3. On the Junos device, define a unique identifier for the client.

```
[edit system services netconf tls]
user@host# edit client-identity client1
```

4. Configure the client's certificate fingerprint in x509c2n:tls-fingerprint format.

   The fingerprint's first octet is the hashing algorithm identifier, and the remaining octets are the result of the hashing algorithm.

```
[edit system services netconf tls client-identity client1]
user@host# set fingerprint
04:D2:96:AF:89:AB:33:A4:F9:5C:0F:34:9E:FC:67:2D:98:C6:08:9B:E8:6C:DE:63:60:1C:F6:CD:1A:43:5A:3
0:AD
```

5. Configure the map type that defines how the server derives the NETCONF username for that client.

```
[edit system services netconf tls client-identity client1]
user@host# set map-type (san-dirname-cn | specified)
```

6. If the map type is `specified`, configure the NETCONF username to use for that client.

```
[edit system services netconf tls client-identity client1]
user@host# set username netconf-user
```

7. Commit the configuration.

```
[edit system services netconf tls client-identity client1]
user@host# commit
```

Configure the Default NETCONF Username Mapping

You can define a default mapping that is used to derive the NETCONF username when a client does not match a client configured at the `[edit system services netconf tls client-identity]` hierarchy level.

To define the default mapping to derive the NETCONF username:

1. Configure the default map type that the server uses to derive the NETCONF username.

```
[edit system services netconf tls]
user@host# set default-client-identity map-type (san-dirname-cn | specified)
```

2. If the map type is `specified`, configure the default NETCONF username.

```
[edit system services netconf tls]
user@host# set default-client-identity username netconf-default-user
```

3. Commit the configuration.

```
[edit system services netconf tls]
user@host# commit
```

Configure the User Account for the NETCONF User

When establishing a NETCONF session over TLS, the server maps the client certificate to the NETCONF user that performs the operations on the device for that session. Junos OS supports local users and LDAP remote users for NETCONF-over-TLS sessions. The NETCONF user must either have a user account defined locally on the device, or it must be authenticated by an LDAP server, which then maps it to a local user template account that is defined locally on the device. The following instructions explain how to create a user account on Junos devices.

To create a user account for the NETCONF user on a Junos device:

1. Configure the `user` statement with a unique username, and include the `class` statement to specify a login class that has the permissions required for all actions to be performed by the user.

   For example, the following configuration defines two users: `netconf-user` and `netconf-default-user`.

```
[edit system login]
user@host# set user netconf-user class super-user
user@host# set user netconf-default-user class super-user
```

2. (Optional) Configure the `uid` and `full-name` statements to specify the user's ID and name.

```
[edit system login]
user@host# set user netconf-user uid 2001 full-name "NETCONF TLS User"
```

3. Commit the configuration to activate the user account on the device.

```
[edit]
user@host# commit
```

4. Repeat the preceding steps on each Junos device where the client establishes NETCONF sessions over TLS.

SEE ALSO

Junos OS User Accounts Overview

Configuring Local User Template Accounts for User Authentication

Start the NETCONF-over-TLS Session

The network management system acts as the NETCONF and TLS client. You can use any software for managing the TLS protocol to initiate the NETCONF-over-TLS session with the Junos device.

To start the NETCONF-over-TLS session:

1. Initiate the connection to the NETCONF server on port 6513, and provide the client's certificate and key, the root CA certificate for the server, and all intermediate CA certificates required to validate the client certificate.

```
user@nms:~$ openssl s_client -connect 198.51.100.1:6513 -CAfile all_CAs -cert client.crt -key
client.key -tls1_2
CONNECTED(00000005)
...
[TLS handshake]
...
---
<!-- No zombies were killed during the creation of this user interface -->
<!-- user netconf-user, class j-super-user -->
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file</capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?scheme=http,ftp,file</
capability>
    <capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>35510</session-id>
</hello>
]]>]]>
```

2. Verify that the session maps to the correct NETCONF user.

   The server emits the NETCONF username for that session during the session establishment.

```
<!-- user netconf-user, class j-super-user -->
```

3. Perform NETCONF operations as necessary.

```
<rpc><get-configuration/></rpc>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/20.2R1/junos">
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds="1583544555"
junos:changed-localtime="2020-03-07 01:29:15 UTC">
...
```

4. Close the NETCONF session and TLS connection.

```
<rpc><close-session/></rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/20.2R1/junos">
<ok/>
</rpc-reply>
]]>]]>
<!-- session end at 2020-03-11 19:10:28 UTC -->
closed
```

NETCONF and Shell Sessions over Enhanced Outbound HTTPS

SUMMARY

Client applications can establish Network Configuration Protocol (NETCONF) sessions and shell sessions using enhanced outbound HTTPS on supported Junos devices.

IN THIS SECTION

Understanding NETCONF and Shell Sessions over Enhanced Outbound HTTPS

IN THIS SECTION

Benefits of NETCONF and Shell Sessions over Outbound HTTPS

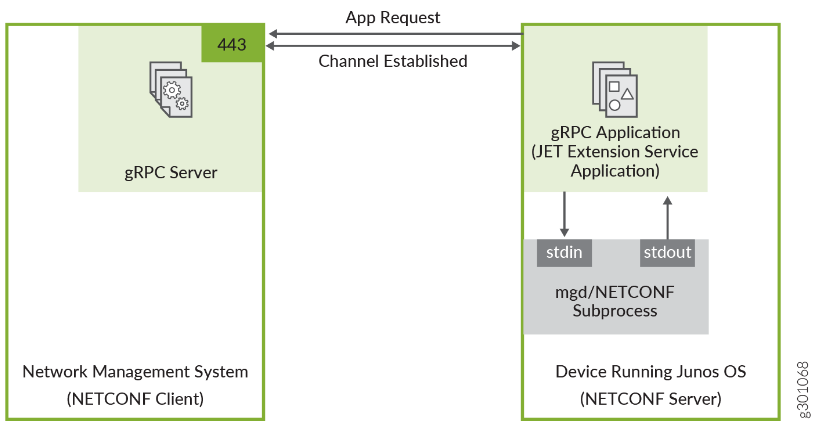
- Enable NETCONF or shell client applications to manage devices that are not accessible through other protocols.
- Enable remote management of devices using certificate-based authentication for the outbound HTTPS client.

NETCONF and Shell Sessions over Outbound HTTPS Overview

You can establish NETCONF and shell sessions over outbound HTTPS between supported Junos devices and a network management system. A NETCONF or shell session over outbound HTTPS enables you to remotely manage devices that might not be accessible through other protocols such as SSH. This might happen, for example, if the device is behind a firewall, and the firewall or another security tool blocks those protocols. HTTPS, on the other hand, uses a standard port, which is typically allowed outbound in most environments.

On supported devices, the Junos software image includes a Juniper Extension Toolkit (JET) application that supports establishing a NETCONF or shell session using outbound HTTPS. The JET application uses the gRPC framework to connect to the outbound HTTPS client, which consists of a gRPC server running on the network management system. gRPC is a language-agnostic, open-source remote procedure call (RPC) framework. Figure 1 on page 69 illustrates the outbound HTTPS setup in its simplest form.

Figure 1: NETCONF and Shell Sessions over Outbound HTTPS


443
App Request
Channel Established
gRPC Server
gRPC Application (JET Extension Service Application)
stdin
stdout
mgd/NETCONF Subprocess
Network Management System (NETCONF Client)
Device Running Junos OS (NETCONF Server)
g301068

In this scenario, the gRPC server acts as the NETCONF/shell client, and the JET application is the gRPC client and NETCONF/shell server. The gRPC server listens for connection requests on the specified port, which defaults to port 443. You configure the JET application as an extension service. The relevant connection and authentication information is passed to the script. While the script runs, it automatically attempts to connect to the gRPC server on the configured host and port.

The JET application and gRPC server establish a persistent HTTPS connection over a TLS-encrypted gRPC session. The JET application authenticates the gRPC server using an X.509 digital certificate, and if the authentication is successful, the requested NETCONF or shell session is established over this connection. The NETCONF operations and shell commands execute under the account privileges of the user configured for the extension service application.

The outbound HTTPS connection uses an X.509 digital certificate to authenticate the gRPC server. A digital certificate is an electronic means for verifying your identity through a trusted third party, known as a certificate authority or certification authority (CA). A certificate authority issues digital certificates, which can be used to establish a secure connection between two endpoints through certificate validation. The X.509 standard defines the format for the certificate. To establish a NETCONF or shell session over outbound HTTPS on supported Junos devices, the gRPC server must have a valid X.509 certificate.

Enhanced outbound HTTPS provides support for:

- Connecting to multiple outbound HTTPS clients

- Configuring multiple backup gRPC servers for each outbound HTTPS client
- Establishing multiple, concurrent NETCONF and shell sessions with a given client
- Authenticating the outbound HTTPS client using self-signed or CA-signed X.509 digital certificates
- Authenticating the Junos device using a shared secret

Connection Workflow for Sessions over Enhanced Outbound HTTPS

In a NETCONF or shell session over outbound HTTPS, the gRPC server running on the network management system acts as the NETCONF/shell client, and the JET application on the Junos device is the gRPC client and NETCONF/shell server. You can configure multiple outbound HTTPS clients, and you can configure one or more backup gRPC servers for each client. The JET application connects to only one gRPC server in the client's server list at any one time.

The gRPC client and server perform the following actions to establish a NETCONF or shell session over outbound HTTPS:

1. The gRPC server listens for incoming connections on the specified port, or if no port is specified, on the default port 443.
2. The gRPC client initiates a TCP/IP connection with the configured gRPC server and port. If you configure an outbound HTTPS client with one or more backup gRPC servers, the gRPC client tries to connect to each server in the list until it establishes a connection.
3. The gRPC client sends a TLS `ClientHello` message to initiate the TLS handshake.
4. The gRPC server sends a `ServerHello` message and its certificate.
5. The gRPC client verifies the identity of the gRPC server.
6. The gRPC client sends the device ID and shared secret configured for that outbound HTTPS client to the gRPC server.
7. The outbound HTTPS client requests a NETCONF or shell session, and the gRPC server uses the device ID and shared secret to authenticate the Junos device. If authentication is successful, the session is established.
8. If a NETCONF session is requested, the server and client exchange NETCONF `<hello>` messages.
9. The NETCONF or shell client application performs operations as needed.

The gRPC client initiates another TCP/IP connection with the same gRPC server, and the gRPC client and server repeat the process, which enables the outbound HTTPS client to establish multiple NETCONF and shell sessions with the network device.

How to Establish NETCONF and Shell Sessions over Enhanced Outbound HTTPS

IN THIS SECTION

You can use the JET application that is included as part of the Junos software image to establish NETCONF and shell sessions over outbound HTTPS between network management systems (NMS) and supported Junos devices. The JET application, configured as an extension service, initiates a connection to a gRPC server running on an NMS and establishes a persistent HTTPS connection over a TLS-encrypted gRPC session. The NETCONF or shell session runs over this HTTPS connection. In this scenario, the gRPC server is the NETCONF/shell client, and the JET application is the gRPC client and NETCONF/shell server.

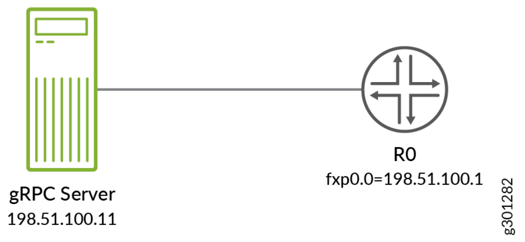
The following hardware and software are required for establishing sessions over enhanced outbound HTTPS:

- Network management system running Python 3.5 or later
- Device running Junos OS Evolved or device running Junos OS with upgraded FreeBSD Release 20.3 or later that also supports running JET applications

NOTE: For supported devices, see Feature Explorer.

Figure 2 on page 72 illustrates the setup referenced in the tasks that follow. The management interface name on the Junos device varies depending on the platform and OS.

Figure 2: NETCONF over Outbound HTTPS Topology



Before the client and server can establish a NETCONF or shell session over outbound HTTPS, you must satisfy the requirements discussed in the following sections:

Obtain an X.509 Certificate for the gRPC Server

The outbound HTTPS connection uses an X.509 public key certificate to authenticate the identity of the gRPC server running on the network management system. The gRPC stack supports the X.509 v3 certificate format.

The requirements for the gRPC server's certificate are:

- The certificate can be self-signed or signed by a certificate authority (CA).
- The certificate must define either the gRPC server's hostname in the Common Name (CN) field, or it must define the gRPC server's IP address in the SubjectAltName (SAN) IP Address field. The Junos device must use the same value to establish the connection to the server. If the certificate defines the SubjectAltName IP Address field, the device ignores the Common Name field during authentication.
- The certificate must be PEM-encoded and use a **.crt** extension.
- The certificate and its key must be named **server.crt** and **server.key**, respectively.

To use OpenSSL to obtain a certificate:

1. Generate a private key, and specify the key length in bits.

```
user@nms:~$ openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
...++++
........................................................................++++
e is 65537 (0x010001)
```

NOTE: We recommend using 3072 bits or greater for the size of the private key. The key length should not exceed 4096 bits.

2. If you are connecting to the gRPC server's IP address, update your **openssl.cnf** or equivalent configuration file to define the `subjectAltName=IP` extension with the gRPC server's address.

```
user@nms:~$ cat openssl.cnf
# OpenSSL configuration file.
...
extensions              = v3_sign
...
[v3_sign]
subjectAltName=IP:198.51.100.11
```

3. Generate a certificate signing request (CSR), which contains the entity's public key and information about their identity.

```
user@nms:~$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Sunnyvale
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Juniper
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:nms.example.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

4. Generate the certificate by doing one of the following:

   - Send the CSR to a certificate authority to request an X.509 certificate, and provide the configuration file to include any additional extensions.

- Sign the CSR with a CA to generate the client certificate, and include the `-extfile` option if you need to reference your configuration file and extensions.

  ```
  user@nms:~$ openssl x509 -req -in server.csr -CA RootCA.crt -CAkey RootCA.key -set_serial
  0101 -out server.crt -days 365 -sha256 -extfile openssl.cnf
  Signature ok
  subject=C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = nms.example.com
  Getting Private key
  ```

- Sign the CSR with the server key to generate a self-signed client certificate, and include the `-extfile` option if you need to reference your configuration file and extensions.

  ```
  user@nms:~$ openssl x509 -req -in server.csr -signkey server.key -out server.crt -days 365
  -sha256 -extfile openssl.cnf
  Signature ok
  subject=C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = nms.example.com
  Getting Private key
  ```

5. Verify that the Common Name (CN) field and extensions, if provided, are correct.

   ```
   user@nms:~$ openssl x509 -text -noout -in server.crt
   Certificate:
       Data:
           Version: 3 (0x2)
           ...
           Subject: C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = nms.example.com
           ...
           X509v3 extensions:
               X509v3 Subject Alternative Name:
                   IP Address:198.51.100.11
       ...
   ```

Set Up the gRPC Server

The network management system requires the following software:

- Python 3.5 or later

The network management system and the JET application on the Junos device use the gRPC framework to establish a persistent HTTPS connection over a TLS-encrypted gRPC session. The network management system must have the gRPC stack installed and run a gRPC server that listens on the specified port for the connection request. Juniper Networks provides the necessary proto definition files and sample gRPC server application files in the Juniper Networks `netconf-https-outbound` repository on GitHub.

This section sets up the gRPC server on a network management system running Ubuntu 18.04. If you are running a different operating system, use the commands appropriate for your OS.

To set up the gRPC server on a network management system running Ubuntu 18.04:

1. Install `pip` for Python 3.

```
user@nms:~$ sudo apt install python3-pip
```

2. Install the `grpcio` package.

```
user@nms:~$ sudo pip3 install grpcio==1.29.0
```

3. Install the `grpcio-tools` package.

```
user@nms:~$ sudo pip3 install grpcio-tools==1.18.0
```

NOTE: If you encounter installation errors for the `grpcio` or `grpcio` packages, try installing the latest version.

4. Go to the Juniper GitHub repository at https://github.com/Juniper/netconf-https-outbound, and select the directory corresponding to the release running on the Junos device.

Release	Directory
Junos OS Release 20.3R1 or later	**20.3**
Junos OS Evolved Release 22.4R1 or later	**junos-evolved/22.4**

5. Download the application and proto files in the GitHub directory to the directory on the network management system where the gRPC server's certificate resides.

   a. Select each file, click the **Raw** button, and copy the URL for the file.

b. Download the file by using the URL with the download tool of your choice, for example, `wget` or `curl`.

```
user@nms:~$ ls
jnx_common_base_types.proto  jnx_netconf_service.proto  nc_grpc_server.py
request_session.py  server.crt
```

6. Use the protocol buffer compiler, `protoc`, to compile each proto definition file and generate Python code, which produces two output files for each proto file.

```
user@nms:~$ python3 -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
filename.proto
```

For example:

```
user@nms:~$ python3 -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
jnx_common_base_types.proto
user@nms:~$ python3 -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
jnx_netconf_service.proto

user@nms:~$ ls jnx*.py
jnx_common_base_types_pb2_grpc.py  jnx_netconf_service_pb2_grpc.py
jnx_common_base_types_pb2.py       jnx_netconf_service_pb2.py
```

7. Start the gRPC server, and specify the port for the connection, if it's different from the default port 443.

```
user@nms:~$ python3 nc_grpc_server.py -p 50051
```

NOTE: You might need to execute the script with root permissions to listen on port 443.

The gRPC server listens indefinitely on the specified port for incoming connections. After you configure the Junos device to connect to the gRPC server and a connection and session are established, you can perform NETCONF operations or shell commands as appropriate.

Configure the User Account for the NETCONF or Shell User

To establish a NETCONF or shell session over outbound HTTPS, you must create a user account locally on the Junos device. You use this account to perform the NETCONF or shell operations on the device for that session. The JET application runs using the permissions configured for this account.

To create a user account on a Junos device:

1. Configure the `user` statement with a unique username, and include the `class` statement to specify a login class that has the permissions required for all actions to be performed by the user. For example:

```
[edit system login]
user@R0# set user netconf-user class super-user
```

2. (Optional) Configure the `uid` and `full-name` statements to specify a unique user ID and the user's name.

```
[edit system login]
user@R0# set user netconf-user uid 2001 full-name "NETCONF User"
```

3. Commit the configuration to activate the user account on the device.

```
[edit system login]
user@R0# commit
```

4. Repeat the preceding steps on each Junos device where the client needs to establish NETCONF or shell sessions over outbound HTTPS.

Configure the Outbound HTTPS Clients

Enhanced outbound HTTPS enables you to configure multiple outbound HTTPS clients at the `[edit system services outbound-https]` hierarchy level and configure multiple backup gRPC servers for each client. The JET application connects to only one gRPC server in the client's server list at any one time.

Before you configure the device, you will need the following information:

- The port on which the gRPC server is listening for connections.
- The contents of the SubjectAltName IP Address field, or if there is no such field, the contents of the Common Name (CN) field in the gRPC server's certificate.
- The contents of the gRPC server's certificate, if it's self-signed, or the contents of the CA certificates, if the server certificate is authenticated using a certificate chain.

To configure an outbound HTTPS client:

1. Navigate to the outbound HTTPS client hierarchy, and define an identifier that uniquely identifies the outbound HTTPS client.

```
[edit]
user@R0# edit system services outbound-https client nms1
```

2. Define the device identifier, which is a user-defined string that the gRPC server uses to identify and authenticate the Junos device during session establishment.

```
[edit system services outbound-https client nms1]
user@R0# set device-id router1
```

3. Define a shared secret string, which is a user-defined string that the gRPC server uses to authenticate the Junos device during session establishment.

```
[edit system services outbound-https client nms1]
user@R0# set secret my-shared-secret
```

   The device stores the shared secret string as an encrypted value in the configuration database.

```
[edit system services outbound-https client nms1]
user@R0# show secret
secret "$9$atZjq36ABIE/CIcyr8LGDik.53nCO1R690IcSMWJGDikPz39"; ## SECRET-DATA
```

4. (Optional) Define the method used to reestablish a disconnected outbound HTTPS connection as `sticky` or `in-order`.

```
[edit system services outbound-https client nms1]
user@R0# set reconnect-strategy sticky
```

5. (Optional) Define the time in seconds that the gRPC client waits in between attempts to connect to the outbound HTTPS client's list of servers.

```
[edit system services outbound-https client nms1]
user@R0# set waittime 30
```

6. Configure the hostname or IPv4 address for one or more gRPC servers and the port on which the server is listening for outbound HTTPS connection requests.

The hostname or IP address must match the value of the Common Name (CN) field or the SubjectAltName IP Address field, respectively, in that gRPC server's certificate.

```
[edit system services outbound-https client nms1]
user@R0# set 198.51.100.11 port 50051
```

7. For each gRPC server, configure the `trusted_cert` statement with the certificate information required to authenticate the server.

   - If the server's certificate is self-signed, configure the contents of the gRPC server's certificate, **server.crt**, omitting any newlines.

     ```
     [edit system services outbound-https client nms1]
     user@R0# set 198.51.100.11 trusted-cert "-----BEGIN CERTIFICATE-----MIIFH***FjQ==-----END
     CERTIFICATE-----"
     ```

   - If the server's certificate is authenticated using a certificate chain, concatenate any intermediate CA and root CA certificates in that order, remove all newlines, and configure the resulting single string.

     ```
     [edit system services outbound-https client nms1]
     user@R0# set 198.51.100.11 trusted-cert "-----BEGIN CERTIFICATE-----MIIFA***ioUS-----END
     CERTIFICATE----------BEGIN CERTIFICATE-----MIIFX***0xUc=-----END CERTIFICATE-----"
     ```

   NOTE: To easily generate the value for the `trusted_cert` statement, you can concatenate the appropriate certificates in the required order and remove any newlines, for example, by using a command similar to the following:

   ```
   user@nms:~$ cat IntermediateCA.crt RootCA.crt | tr -d '\n' > allCA
   ```

8. Repeat the preceding steps for each outbound HTTPS client that will manage the Junos device.

9. Commit the configuration.

```
[edit system services outbound-https client nms1]
user@R0# commit and-quit
```

NOTE: If the outbound HTTPS extension service is already running, and you add, delete, or modify an outbound HTTPS client and commit the configuration, you do not need to restart the service for the changes to take effect. They are picked up automatically.

Configure the Outbound HTTPS Extension Service on Junos Devices

Junos releases that support NETCONF and shell sessions over outbound HTTPS include a JET application and supporting files in the software image. Table 5 on page 80 outlines the files, which are located in the **/var/db/scripts/jet** directory on the device.

Table 5: JET Files for Sessions over Enhanced Outbound HTTPS

File	Description
nc_grpc_app.pyc	JET application that uses the gRPC framework to establish a persistent HTTPS connection with a gRPC server running on the network management system.
nc_grpc_app_lib.pyc	Required libraries

To configure the Junos device for sessions over outbound HTTPS:

1. Verify that the JET application and related files are present on the device.

```
user@R0> file list /var/db/scripts/jet/nc*.pyc
/var/db/scripts/jet/nc_grpc_app.pyc@ -> /packages/mnt/junos-runtime/var/db/scripts/jet/
nc_grpc_app.pyc
/var/db/scripts/jet/nc_grpc_app_lib.pyc@ -> /packages/mnt/junos-runtime/var/db/scripts/jet/
nc_grpc_app_lib.pyc
```

2. Enter configuration mode.

```
user@R0> configure
Entering configuration mode
```

3. Enable the device to run unsigned Python 3 applications.

```
[edit]
user@R0# set system scripts language python3
```

4. Configure extension service notifications for the loopback address.

```
[edit]
user@R0# set interfaces lo0 unit 0 family inet address 127.0.0.1
user@R0# set system commit notification configuration-diff-format xml
user@R0# set system services extension-service notification allow-clients address 127.0.0.1
```

5. Navigate to the hierarchy of the extension service application.

```
[edit]
user@R0# edit system extensions extension-service application file nc_grpc_app.pyc
```

6. Configure the application to run in the background as a daemonized process.

```
[edit system extensions extension-service application file nc_grpc_app.pyc]
user@R0# set daemonize
```

7. Configure the application to respawn on normal exit.

```
[edit system extensions extension-service application file nc_grpc_app.pyc]
user@R0# set respawn-on-normal-exit
```

8. Configure the usename under whose privileges the application executes and the NETCONF operations and shell commands are performed.

```
[edit system extensions extension-service application file nc_grpc_app.pyc]
user@R0# set username netconf-user
```

9. Commit the configuration.

```
[edit system extensions extension-service application file nc_grpc_app.pyc]
user@R0# commit and-quit
```

   When you commit the configuration, the `daemonize` option causes the application to start automatically.

10. Verify that the application is running.

```
user@R0> show extension-service status nc_grpc_app.pyc
Extension service application details:
Name : nc_grpc_app
Process-id: 81383
Stack-Segment-Size: 16777216B
Data-Segment-Size: 134217728B
```

After the application successfully starts, it logs messages to the **outbound_https.log** file.

NOTE: If the application does not automatically start after you commit the configuration, review the log messages related to this application to troubleshoot the issue. In Junos OS, issue the `show log jet.log` command. In Junos OS Evolved, issue the `show trace application cscript` and `show log messages` commands.

Start the NETCONF or Shell Session

The gRPC server running on the network management system acts as the NETCONF/shell client, and the JET application on the Junos device acts as the gRPC client and NETCONF/shell server. After you start the gRPC server and JET application, the JET application attempts to connect to the gRPC server on the specified port. If the connection is successful, the gRPC client authenticates the gRPC server. If the server authentication is successful, you can then request one or more NETCONF or shell sessions.

Before you begin, you will need the following information:

- The device identifier and shared secret string configured for the outbound HTTPS client

To establish a NETCONF or shell session over enhanced outbound HTTPS:

1. On the network management system, if you did not already start the gPRC server, start the server, and specify the port for the connection.

```
user@nms:~$ python3 nc_grpc_server.py -p 50051
2020-08-03 13:45:52,278 [INFO ]  /home/user/
2020-08-03 13:45:52,279 [INFO ]  first parent process is exited
2020-08-03 13:45:52,287 [INFO ]  second parent process is exited
```

2. To establish one or more sessions with a Junos device, execute the **request_session.py** script. Specify the session type as well as the device ID and shared secret that you configured for that outbound HTTPS client on the Junos device. For example:

- To request a csh session, which is the default, you do not need to specify a session type.

```
user@nms:~$ python3 request_session.py -d router1 -sk my-shared-secret
```

- To request a NETCONF session, include the `-s netconf` option.

```
user@nms:~$ python3 request_session.py -d router1 -sk my-shared-secret -s netconf
```

If the server successfully authenticates the Junos device, the requested session starts.

3. Verify that the session is established by reviewing the output.

- Shell sessions should display the `csh session is started` output, for example:

```
$
 csh session is started
whoami
 netconf-user
ls
  base-config.conf
```

- NETCONF sessions should display the NETCONF capabilities as shown here:

```
<!-- No zombies were killed during the creation of this user interface -->
<!-- user netconf-user, class j-super-user -->
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    ...
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>57602</session-id>
</hello>
]]>]]>
```

4. Perform NETCONF or shell operations as necessary.

```
<rpc><get-configuration/></rpc>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/20.3R1/junos">
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds="1592517292"
junos:changed-localtime="2020-10-18 14:54:52 PDT">
...
</configuration>
</rpc-reply>
]]>]]>
```

5. When you are finished with the session, type `Ctrl+C`.

```
^CForce exit
Killed
```

6. When you are finished using the outbound HTTPS connection, you can stop the extension service application on the Junos device by deleting or deactivating the relevant hierarchy in the configuration and then committing the change.

```
user@R0# delete system extensions extension-service application file nc_grpc_app.pyc
user@R0# commit and-quit
```

NETCONF Sessions over Outbound HTTPS

SUMMARY

Client applications can establish Network Configuration Protocol (NETCONF) sessions using outbound HTTPS on supported devices running Junos OS Release 20.2.

IN THIS SECTION

This topic discusses how to establish NETCONF sessions using outbound HTTPS on devices running Junos OS Release 20.2. For information about establishing NETCONF and shell sessions using enhanced outbound HTTPS, see "NETCONF and Shell Sessions over Enhanced Outbound HTTPS" on page 68.

Understanding NETCONF Sessions over Outbound HTTPS

IN THIS SECTION

Benefits of NETCONF Sessions over Outbound HTTPS

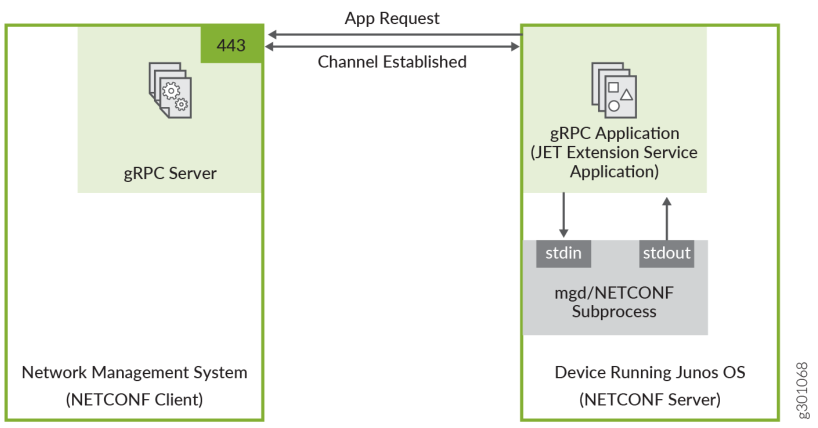
- Enable NETCONF client applications to manage devices that are not accessible through other protocols.
- Enable remote management of devices using certificate-based authentication for the outbound HTTPS client.

NETCONF Sessions over Outbound HTTPS Overview

You can establish NETCONF sessions over outbound HTTPS between supported Junos devices and a network management system. A NETCONF session over outbound HTTPS enables you to remotely manage devices that might not be accessible through other protocols such as SSH. This might happen, for example, if the device is behind a firewall, and the firewall or another security tool blocks those protocols. HTTPS, on the other hand, uses a standard port, which is typically allowed outbound in most environments.

On supported devices, Junos OS includes a Juniper Extension Toolkit (JET) application that supports establishing a NETCONF session using outbound HTTPS. The JET application uses the gRPC framework to connect to the outbound HTTPS client, which consists of a gRPC server running on the network management system. gRPC is a language-agnostic, open-source remote procedure call (RPC) framework. Figure 3 on page 86 illustrates the outbound HTTPS setup in its simplest form.

Figure 3: NETCONF Sessions over Outbound HTTPS



In this scenario, the gRPC server acts as the NETCONF client, and the JET application is the gRPC client and NETCONF server. The gRPC server listens for connection requests on the specified port, which defaults to port 443. You configure the JET application as an extension service. The relevant connection and authentication information is passed to the script. While the script runs, it automatically attempts to connect to the gRPC server on the configured host and port.

The JET application and gRPC server establish a persistent HTTPS connection over a TLS-encrypted gRPC session. The JET application authenticates the gRPC server using an X.509 digital certificate, and if the authentication is successful, the requested NETCONF session is established over this connection. The NETCONF operations execute under the account privileges of the user configured for the extension service application.

The outbound HTTPS connection uses an X.509 digital certificate to authenticate the gRPC server. A digital certificate is an electronic means for verifying your identity through a trusted third party, known as a certificate authority or certification authority (CA). A certificate authority issues digital certificates, which can be used to establish a secure connection between two endpoints through certificate validation. The X.509 standard defines the format for the certificate. To establish a NETCONF session over outbound HTTPS on supported Junos devices, the gRPC server must have a valid X.509 certificate.

The basic outbound HTTPS feature provides support for connecting to a single outbound HTTPS client and configuring one gRPC server for that client. Server authentication must use a self-signed X.509 certificate. You can establish a single NETCONF session over the connection.

Connection Workflow for Sessions over Outbound HTTPS

In a NETCONF session over outbound HTTPS, the gRPC server running on the network management system acts as the NETCONF client, and the JET application on the Junos device is the gRPC client and NETCONF server.

The gRPC client and server perform the following actions to establish a NETCONF session over outbound HTTPS:

1. The gRPC server listens for incoming connections on the specified port, or if no port is specified, on the default port 443.
2. The gRPC client initiates a TCP/IP connection with the configured gRPC server and port.
3. The gRPC client sends a TLS `ClientHello` message to initiate the TLS handshake.
4. The gRPC server sends a `ServerHello` message and its certificate.
5. The gRPC client verifies the identity of the gRPC server.
6. The NETCONF session is established.
7. The server and client exchange NETCONF `<hello>` messages.
8. The NETCONF client application performs operations as needed.

How to Establish a NETCONF Session over Outbound HTTPS

IN THIS SECTION

You can use the JET application that is included as part of the Junos software image to establish a NETCONF session over outbound HTTPS between a network management system (NMS) and supported Junos devices. The JET application, configured as an extension service, initiates a connection to a gRPC server running on an NMS and establishes a persistent HTTPS connection over a TLS-encrypted gRPC session. The NETCONF session runs over this HTTPS connection. In this scenario, the gRPC server is the NETCONF client, and the JET application is the gRPC client and NETCONF server.

The following hardware and software are required for establishing a NETCONF session over outbound HTTPS:

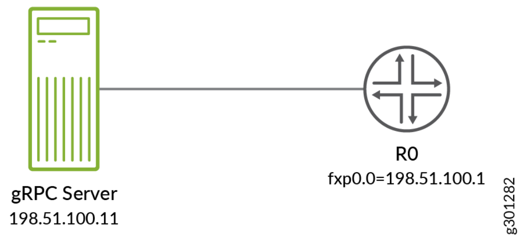
- Network management system running Python 3.5 or later

- Device running Junos OS with upgraded FreeBSD Release 20.2 that also supports running JET applications

NOTE: For supported devices, see Feature Explorer NETCONF sessions over outbound HTTPS.

Figure 4 on page 88 illustrates the setup referenced in the tasks that follow.

Figure 4: NETCONF over Outbound HTTPS Topology



Before the client and server can establish a NETCONF session over outbound HTTPS, you must satisfy the requirements discussed in the following sections:

Obtain an X.509 Certificate for the gRPC Server

The outbound HTTPS connection uses an X.509 public key certificate to authenticate the identity of the gRPC server running on the network management system. The gRPC stack supports the X.509 v3 certificate format.

The requirements for the gRPC server's certificate are:

- The certificate must be self-signed.
- The certificate must define either the gRPC server's hostname in the Common Name (CN) field, or it must define the gRPC server's IP address in the SubjectAltName (SAN) IP Address field. The Junos device must use the same value to establish the connection to the server. If the certificate defines the SubjectAltName IP Address field, the device ignores the Common Name field during authentication.
- The certificate must be PEM-encoded and use a **.crt** extension.
- The certificate and its key must be named **server.crt** and **server.key**, respectively.

To use OpenSSL to obtain a certificate:

1. Generate a private key, and specify the key length in bits.

```
user@nms:~$ openssl genrsa -out server.key 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
...++++
...........................................................................++++
e is 65537 (0x010001)
```

NOTE: We recommend using 3072 bits or greater for the size of the private key.

2. If you are connecting to the gRPC server's IP address, update your **openssl.cnf** or equivalent configuration file to define the `subjectAltName=IP` extension with the gRPC server's address.

```
user@nms:~$ cat openssl.cnf
# OpenSSL configuration file.
...
extensions              = v3_sign
...
[v3_sign]
subjectAltName=IP:198.51.100.11
```

3. Generate a certificate signing request (CSR), which contains the client's public key and information about their identity.

```
user@nms:~$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Sunnyvale
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Juniper
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:nms.example.com
Email Address []:
Please enter the following 'extra' attributes
```

```
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

4. Generate the certificate.

   Sign the CSR with the server key to generate a self-signed client certificate, and include the `-extfile` option if you need to reference your configuration file and extensions.

```
user@nms:~$ openssl x509 -req -in server.csr -signkey server.key -out server.crt -days 365 -
sha256 -extfile openssl.cnf
Signature ok
subject=C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = nms.example.com
Getting Private key
```

5. Verify that the Common Name (CN) field and extensions, if provided, are correct.

```
user@nms:~$ openssl x509 -text -noout -in server.crt
Certificate:
    Data:
        Version: 3 (0x2)
        ...
        Subject: C = US, ST = CA, L = Sunnyvale, O = Juniper, CN = nms.example.com
        ...
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                IP Address:198.51.100.11
    ...
```

6. (Optional) Copy the **server.crt** file to the **/var/db/scripts/jet** directory on the device running Junos OS to use the certificate file for authentication.

```
user@nms:~$ scp server.crt <device-hostname-or-ip>:/var/db/scripts/jet

Password:
server.crt                                                                    100%
1862     3.9MB/s   00:00
```

NOTE: You can omit this step if the key size is less than or equal to 4096 bits and you instead configure the certificate's contents in the JET application's `trusted_certs` argument on the Junos device.

Set Up the gRPC Server

The network management system requires the following software:

- Python 3.5 or later

The network management system and the JET application on the Junos device use the gRPC framework to establish a persistent HTTPS connection over a TLS-encrypted gRPC session. The network management system must have the gRPC stack installed and run a gRPC server that listens on the specified port for the connection request. Juniper Networks provides the necessary proto definition files and sample gRPC server application files in the Juniper Networks `netconf-https-outbound` repository on GitHub.

This section sets up the gRPC server on a network management system running Ubuntu 18.04. If you are running a different operating system, use the commands appropriate for your OS.

To set up the gRPC server on a network management system running Ubuntu 18.04:

1. Install `pip` for Python 3.

```
user@nms:~$ sudo apt install python3-pip
```

2. Install the `grpcio` package.

```
user@nms:~$ sudo pip3 install grpcio==1.29.0
```

3. Install the `grpcio-tools` package.

```
user@nms:~$ sudo pip3 install grpcio-tools==1.18.0
```

4. Go to the Juniper GitHub repository at https://github.com/Juniper/netconf-https-outbound, and select the directory corresponding to the release running on the Junos device.
5. Download the application and proto files in the GitHub directory to the directory on the network management system where the gRPC server's certificate resides.
   a. Select each file, click the **Raw** button, and copy the URL for the file.

b. Download the file by using the URL with the download tool of your choice, for example, `wget` or `curl`.

```
user@nms:~$ ls
nc_grpc.proto  nc_grpc_server.py  server.crt
```

6. Use the protocol buffer compiler, `protoc`, to compile each proto definition file and generate Python code, which produces two output files for each proto file.

```
user@nms:~$ python3 -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
filename.proto
```

For example:

```
user@nms:~$ python3 -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=. nc_grpc.proto
```

7. Start the gRPC server, and specify the port for the connection, if it's different from the default port 443.

```
user@nms:~$ python3 nc_grpc_server.py -p 50051
```

NOTE: You might need to execute the script with root permissions to listen on port 443.

The gRPC server listens indefinitely on the specified port for incoming connections. After you configure the Junos device to connect to the gRPC server and a connection and session are established, you can perform NETCONF operations as appropriate.

Configure the User Account for the NETCONF User

To establish a NETCONF session over outbound HTTPS, you must create a user account locally on the Junos device. You use this account to perform the NETCONF operations on the device for that session. The JET application runs using the permissions configured for this account.

To create a user account on a Junos device:

1. Configure the `user` statement with a unique username, and include the `class` statement to specify a login class that has the permissions required for all actions to be performed by the user. For example:

```
[edit system login]
user@R0# set user netconf-user class super-user
```

2. (Optional) Configure the `uid` and `full-name` statements to specify a unique user ID and the user's name.

```
[edit system login]
user@R0# set user netconf-user uid 2001 full-name "NETCONF User"
```

3. Commit the configuration to activate the user account on the device.

```
[edit system login]
user@R0# commit
```

4. Repeat the preceding steps on each Junos device where the client needs to establish NETCONF sessions over outbound HTTPS.

Configure the Outbound HTTPS Client

The JET application can connect to only one outbound HTTPS client. You configure the connection and authentication information for the client as command-line arguments to the JET script. Table 6 on page 93 outlines the arguments.

Table 6: nc_grpc_app.py Arguments

Argument	Value
`--device` or `-d`	The hostname or IPv4 address of the gRPC server to which the JET application connects. The argument value must match the hostname in the Common Name (CN) field or the IP address in the SubjectAltName IP address field in the gRPC server's certificate.
`--port` or `-p`	(Optional) Port on which the JET application attempts to connect to the gRPC server. Omit this argument to use the default port 443.

Table 6: nc_grpc_app.py Arguments *(Continued)*

Argument	Value
`--trusted_certs` or `-ts`	(Optional) The gRPC server's certificate contents between the `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` lines, omitting any newlines. You can omit this argument if you instead copy the certificate to the **/var/db/scripts/jet** directory on the device. You must copy the certificate to the device for key sizes greater than 4096 bits.

Before you begin, you will need the values for the script arguments, including:

- The port on which the gRPC server is listening for connections.
- The contents of the SubjectAltName IP Address field, or if there is no such field, the contents of the Common Name (CN) field in the gRPC server's certificate.
- The contents of the gRPC server's certificate between `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----`, omitting any newlines. This information is only required when you configure the certificate contents as a script argument instead of copying the certificate to the device running Junos OS.

To configure the outbound HTTPS client:

1. Navigate to the hierarchy of the **nc_grpc_app.py** extension service application.

```
[edit]
user@R0# edit system extensions extension-service application file nc_grpc_app.py
```

2. Configure the arguments that are passed to the application when it starts.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# set arguments "--device 198.51.100.11 --port 50051 --trusted_certs MIIFR***fhd7y"
```

3. Commit the configuration.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# commit
```

Configure the Outbound HTTPS Extension Service on Junos Devices

Junos releases that support NETCONF sessions over outbound HTTPS include a JET application and supporting files in the software image. Table 7 on page 95 outlines the files, which are located in the **/var/db/scripts/jet** directory on the device.

Table 7: JET Files for Sessions over Outbound HTTPS

Files	Description
nc_grpc_app.py	JET application that uses the gRPC framework to establish a persistent HTTPS connection with a gRPC server running on the network management system.
nc_grpc_pb2.py **nc_grpc_pb2_grpc.py**	Required libraries

To configure the Junos device for sessions over outbound HTTPS:

1. Verify that the JET application and related files are present on the device.

```
user@R0> file list /var/db/scripts/jet/nc*
/var/db/scripts/jet/nc_grpc_app.py@ -> /packages/mnt/junos-runtime/var/db/scripts/jet/
nc_grpc_app.py
/var/db/scripts/jet/nc_grpc_pb2.py@ -> /packages/mnt/junos-runtime/var/db/scripts/jet/
nc_grpc_pb2.py
/var/db/scripts/jet/nc_grpc_pb2_grpc.py@ -> /packages/mnt/junos-runtime/var/db/scripts/jet/
nc_grpc_pb2_grpc.py
```

2. Enter configuration mode.

```
user@R0> configure
Entering configuration mode
```

3. Enable the device to run unsigned Python 3 applications.

```
[edit]
user@R0# set system scripts language python3
```

4. Navigate to the hierarchy of the extension service application.

```
[edit]
user@R0# edit system extensions extension-service application file nc_grpc_app.py
```

5. Configure the application to run in the background as a daemonized process.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# set daemonize
```

6. Configure the application to respawn on normal exit.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# set respawn-on-normal-exit
```

7. Configure the usename under whose privileges the application executes and the NETCONF operations are performed.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# set username netconf-user
```

8. Commit the configuration.

```
[edit system extensions extension-service application file nc_grpc_app.py]
user@R0# commit and-quit
```

   When you commit the configuration, the `daemonize` option causes the application to start automatically.

9. Verify that the application is running.

```
user@R0> show extension-service status nc_grpc_app.py
Extension service application details:
Name : nc_grpc_app
Arguments: -device 198.51.100.11 -port 50051 -trusted_certs *****
Stack-Segment-Size: 16777216B
Data-Segment-Size: 0B
```

After the application successfully starts, it logs messages to the **outbound_https.log** file.

NOTE: If the application does not automatically start after you commit the configuration, review the log messages related to this application to troubleshoot the issue. In Junos OS, issue the `show log jet.log` command.

Start the NETCONF Session

The gRPC server running on the network management system acts as the NETCONF client, and the JET application on the Junos device acts as the gRPC client and NETCONF server. After you start the gRPC server and JET application, the JET application attempts to connect to the gRPC server on the specified port. If the connection is successful, the gRPC client authenticates the gRPC server. If the server authentication is successful, the NETCONF session starts automatically.

To establish a NETCONF session over outbound HTTPS:

1. On the network management system, if you did not already start the gPRC server, start the server, and specify the port for the connection.

   ```
   user@nms:~$ python3 nc_grpc_server.py -p 50051
   server started
   ```

   The NETCONF session starts automatically.

2. Verify that the session is successfully established by reviewing the output.

   NETCONF sessions should display the NETCONF capabilities as shown here:

   ```
   Initial hand shake completed and the client is trusted
   <!-- No zombies were killed during the creation of this user interface -->
   <!-- user netconf-user, class j-super-user -->
   <hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
     <capabilities>
       <capability>urn:ietf:params:netconf:base:1.0</capability>
       <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
       ...
       <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
       <capability>http://xml.juniper.net/dmi/system/1.0</capability>
     </capabilities>
     <session-id>57602</session-id>
   </hello>
   ]]>]]>
   ```

3. Perform NETCONF operations as necessary.

```
<rpc><get-configuration/></rpc>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/20.2R1/junos">
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds="1592517292"
junos:changed-localtime="2020-09-18 14:54:52 PDT">
...
</configuration>
</rpc-reply>
]]>]]>
```

4. When you are finished with the session, type `Ctrl+C`.

```
^CForce exit
Killed
```

5. When you are finished using the outbound HTTPS connection, you can stop the extension service application on the Junos device by deleting or deactivating the relevant hierarchy in the configuration and then committing the change.

```
user@R0# delete system extensions extension-service application file nc_grpc_app.py
user@R0# commit and-quit
```

NETCONF Call Home Sessions

SUMMARY

NETCONF Call Home enables a NETCONF server to initiate a secure connection to a NETCONF client. The NETCONF client can then establish an SSH session and NETCONF session with the server.

IN THIS SECTION

NETCONF Call Home enables a NETCONF client to remotely manage Junos devices in cases where the client cannot initiate a connection to the network device. The Junos device that is acting as the NETCONF server initiates and maintains secure connections to a predefined set of NETCONF clients. The following sections discuss the NETCONF Call Home feature.

Understanding NETCONF Call Home

IN THIS SECTION

NETCONF Call Home Overview

For normal NETCONF sessions over SSH, the NETCONF client first initiates a TCP connection to the Junos device that is acting as the NETCONF server. The client then starts the SSH session followed by the NETCONF session. In a NETCONF call home scenario, the roles of the TCP server and client are reversed. The NETCONF server initiates the TCP connection to the NETCONF client. After the connection is established, the NETCONF client then starts the SSH session and NETCONF session as normal.

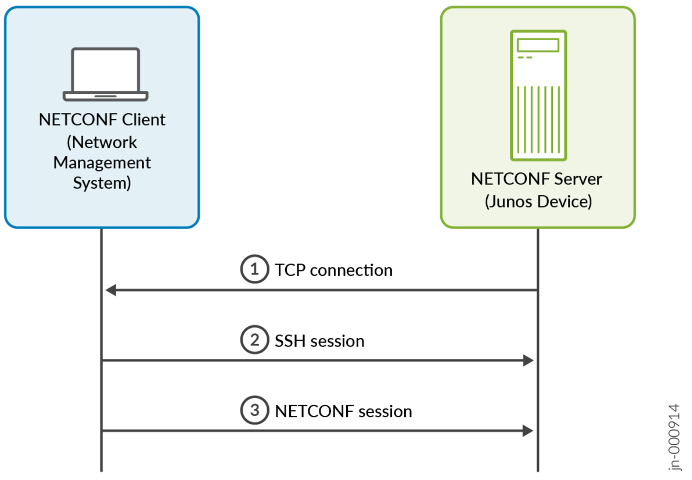
NETCONF Call Home enables you to remotely manage Junos devices that might not be otherwise accessible. This might happen, for example, if the device is behind a firewall, and the firewall or another security tool restricts management access to the device or implements Network Address Translation (NAT). In such cases, the NETCONF client might not be able to initiate the connection with the Junos device. However, you can configure the Junos device to initiate, establish, and maintain a connection with a predefined set of network management systems.

You can use NETCONF Call Home in the following scenarios:

- Initial deployment
- General device management

Figure 5 on page 100 outlines the call home sequence. In a NETCONF call home scenario, the network management system (NMS) is the TCP server, and the Junos device is the TCP client. You configure the TCP server to listen for connection requests on a specified port. The NETCONF Call Home standard port is 4334, but you can configure any valid port number that does not conflict with another service. You configure the Junos device to connect to one or more predefined network management systems on the specified port. After the devices establish a TCP connection, the NMS assumes its role as SSH client and NETCONF client.

Figure 5: NETCONF Call Home Connection


NETCONF Client (Network Management System)
NETCONF Server (Junos Device)
1 TCP connection
2 SSH session
3 NETCONF session
jn-000914

The NETCONF client initiates an SSH session over the TCP connection. During SSH session establishment, the NETCONF client must authenticate the server by validating the server's presented host key or certificate. The NETCONF server must also authenticate the client. The client can choose any SSH authentication method supported by the server. The server first checks certificate-based authentication, then key-based authentication, and finally password-based authentication.

We recommend using SSH certificate-based authentication for users and hosts. Certificate-based authentication enables you to set up SSH access to a device with password-less login for users and gives the capability to trust hosts without the need to verify key fingerprints. Additionally, you can define the length of time that a given certificate is valid. For certificate-based authentication, the server and client each send their SSH certificate. You must configure the client with the CA public key that verifies the server's certificate. Similarly, you must configure the server with the CA public key that verifies the client's certificate.

NOTE: Junos OS includes a customized implementation of OpenSSH for device management. Security fixes are backported as needed, independent of the OpenSSH version numbers. The version displayed in CLI output (for example, `show version`) may not reflect all applied patches. Always refer to Juniper Security Advisories (JSAs) for vulnerability impact assessments.

For more information about NETCONF Call Home, see RFC 8071, *NETCONF Call Home and RESTCONF Call Home*.

Benefits of NETCONF Call Home

- NETCONF Call Home can streamline the initial deployment of devices.
- NETCONF Call Home using certificate-based authentication can simplify the management and scaling of authentication keys and offer stronger security compared to traditional password-based approaches.

How to Set Up NETCONF Call Home

IN THIS SECTION

To configure NETCONF Call Home, perform the steps described in the following sections:

Configure the Junos User Account

For all SSH authentication methods, the NETCONF client needs a local user account or user template account on each device where it establishes a NETCONF session.

To create a local user template account:

1. Configure the `user` statement at the `[edit system login]` hierarchy level and specify a username. Include the `class` statement, and specify a login class that has the permissions required for all actions to be performed by the user.

   ```
   [edit system login]
   user@host# set user username class class-name
   ```

2. Commit the configuration

   ```
   [edit]
   user@host# commit
   ```

3. Repeat the preceding steps on each Junos device where the client application establishes NETCONF sessions.

Configure SSH Authentication

IN THIS SECTION

During SSH session establishment, the NETCONF client authenticates the server by validating the server's presented host key or certificate. Similarly, the NETCONF server authenticates the client. The server first checks certificate-based authentication, then key-based authentication, and finally password-based authentication. You must configure the server and client for whichever authentication method you choose. The SSH authentication methods are described in the following sections:

Configure Password-Based Authentication

If the NETCONF client uses password-based authentication, configure the Junos user account with a password.

To create a text-based password:

1. Include the `plain-text-password` or `encrypted-password` statement for the user account.

   - To enter a password as text, issue the following command. You are prompted for the password, which is encrypted before being stored.

     ```
     [edit system login user username authentication]
     user@host# set plain-text-password
     New password: password
     Retype new password: password
     ```

To store a pre-encrypted password, for example, a password that you previously created and hashed using Message Digest 5 (MD5), issue the following command:

```
[edit system login user username authentication]
user@host# set encrypted-password "password"
```

2. Commit the configuration.

```
[edit system login user username authentication]
user@host# commit
```

3. Repeat the preceding steps on each Junos device where the client application establishes NETCONF sessions using password-based authentication.

Configure Key-Based Authentication

If the NETCONF client uses key-based authentication, first generate the user's SSH key pair on the configuration management server. Then configure the public key under the corresponding user account on the Junos device.

To configure key-based authentication for the NETCONF client:

1. Generate the SSH key pair for the given user.

   On the network management system, issue the `ssh-keygen` command and provide the appropriate arguments.

```
user@nms:~$ ssh-keygen options
```

   For example:

```
netconf-user@nms:~$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/netconf-user/.ssh/id_rsa):
Created directory '/home/netconf-user/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/netconf-user/.ssh/id_rsa
Your public key has been saved in /home/netconf-user/.ssh/id_rsa.pub
...
```

By default, the ssh-keygen command stores the public and private keys in files in the **.ssh** subdirectory of the user home directory; the filename depends on the encoding and SSH version. See the ssh-keygen manual page for more information about the command options.

2. On the Junos device, associate the public key with the Junos login account for that user.

```
[edit system login user username authentication]
user@host# set load-key-file URL
```

Junos OS copies the contents of the specified file onto the device running Junos OS. *URL* is the path to the file that contains one or more public keys. For information about specifying URLs, see the CLI User Guide.

NOTE: Alternatively, you can configure the public key by including the ssh-rsa or equivalent statement at the [edit system login user *username* authentication] hierarchy level.

3. Commit the configuration.

```
[edit]
user@host# commit
```

4. Repeat the steps for configuring the public key on each Junos device where the client application establishes a NETCONF session.

During the initialization of the SSH connection, the client can authenticate the identity of the NETCONF server by using the server's public key. You can either install the server's public key on the network management system, or you can pass in the key during session initialization.

Configure Certificate-Based Authentication

NETCONF Call Home supports using SSH certificates for authentication. An SSH certificate comprises an SSH public key that has been signed by a certificate authority (CA). A CA private key (signing key) signs the host public key to generate the host certificate. Similarly a CA private key signs the user public key to generate the user certificate. The host certificate authenticates the host to users, and the user certificate authenticates the user to hosts.

You can use OpenSSH to create SSH certificates and the associated CA infrastructure. The OpenSSH certificate specifies a public key, identity information, and validity constraints. To use SSH certificates for authentication you must:

- Create the certificate authorities and generate the CA signing keys.

- Generate the user's SSH key pair.
- Generate the user and host certificates by using the CA private key to sign the user public key and host public key.
- Configure the CA public key as trusted on the Junos device and the network management system.
- Configure the authorized principals on the Junos device.

By default, generated certificates are valid for all users. To limit the user certificate to a specific set of users (principals), you must specify the authorized principals when you generate the certificate. Junos devices require certificates that specify one or more authorized principals. You must also configure the list of authorized principals on the Junos device. If *any* authorized principal configured on the device matches an authorized principal in the certificate, the device authenticates the certificate. The connecting user is authorized even if they are not explicitly configured as an authorized principal in the certificate or on the device.

To configure certificate-based authentication, perform the following steps:

Generate the Certificate Authorities

1. On any Linux/FreeBSD system, generate the certificate authority signing keys.

```
user@nms:~$ ssh-keygen -t type -f output-key-file -N passphrase -b bits
```

   For example, the following commands create separate CA signing keys for signing the user and host certificates.

```
user@nms:~$ sudo ssh-keygen -t rsa -f /root/.ssh/user_ca -N "" -b 4096
Generating public/private rsa key pair.
Your identification has been saved in /root/.ssh/user_ca
Your public key has been saved in /root/.ssh/user_ca.pub
The key fingerprint is:
SHA256:0GbR/Lar1eH3/lULO66XSikD+gSwbibgTtFdnHNwlKg root@nms
...
```

```
user@nms:~$ sudo ssh-keygen -t rsa -f /root/.ssh/host_ca -N "" -b 4096
Generating public/private rsa key pair.
Your identification has been saved in /root/.ssh/host_ca
Your public key has been saved in /root/.ssh/host_ca.pub
The key fingerprint is:
```

```
SHA256:Aps80CYHqK8jRaMt2BNkjS1SrJpuEcgF7QTlBoB/ygI root@nms
...
```

2. Configure the network management system to trust the host CA.

   Copy the contents of the host CA public key and add it to the global or user's SSH known hosts file to mark it as a trusted CA. In the following example, we copy the host CA public key contents and add `@cert-authority * host-ca-public-key` to the user's **known_hosts** file.

   ```
   user@nms:~$ sudo cat /root/.ssh/host_ca.pub
   ssh-rsa AAAAB3Nz...
   ```

   ```
   user@nms:~$ cat ~/.ssh/known_hosts
   @cert-authority * ssh-rsa AAAAB3Nz...
   ```

   NOTE: The NMS uses the host CA public key to verify the host certificate. To trust the certificate for only specific hosts or domains, replace the wildcard (*) with the appropriate host or domain names.

3. Copy the user CA public key to the Junos device.

   ```
   user@nms:~$ scp path-to-user-CA-public-key user@device-ip:/var/tmp/
   ```

   For example:

   ```
   user@nms:~$ sudo scp /root/.ssh/user_ca.pub user@198.51.100.1:/var/tmp/
   ```

4. On the Junos device, configure the user CA public key as a trusted CA key.

   ```
   [edit system services ssh]
   user@host# set trusted-user-ca-key-file path-to-user-CA-public-key
   ```

The following example adds the CA public key that is stored in the **user_ca.pub** file as a trusted user CA key.

```
[edit system services ssh]
user@host# set trusted-user-ca-key-file /var/tmp/user_ca.pub
user#host# commit
```

When you configure this statement, the device adds the user CA public key to the **/var/etc/ssh_trusted_user_ca.pub** file. The Junos device uses the user CA public key to verify the user certificate during SSH session establishment.

Generate the User Certificate

To generate the SSH certificate for the NETCONF client:

1. On the client NMS, generate an SSH key pair for the user.

```
user@nms:~$ ssh-keygen -t (rsa | ecdsa | ed25519) -f output-key-file -N passphrase -b bits
```

For example:

```
lab@nms:~$ ssh-keygen -t rsa -f ~/.ssh/id_rsa -N "" -b 4096
Generating public/private rsa key pair.
Your identification has been saved in /home/lab/.ssh/id_rsa
Your public key has been saved in /home/lab/.ssh/id_rsa.pub
...
```

2. Copy the user's public key to the CA server that has the CA signing keys, if the devices are different.

3. Generate the user certificate.

   Use the user CA private key to sign the user's public key, and specify the list of authorized principals (users) that can access the Junos device using this certificate.

```
user@nms:~$ ssh-keygen -s path-to-user-ca-private-key -I key-identity -n
principal1,principal2,principal3 -V validity-interval path-to-user-public-key
```

The following command uses the CA private key stored in **user_ca** to sign the user's public key stored in **id_rsa.pub**. The certificate is valid for two principals: lab and netconf-user.

```
user@nms:~$ sudo ssh-keygen -s /root/.ssh/user_ca -I netconfusers -n lab,netconf-user -V
+52w6d /home/lab/.ssh/id_rsa.pub
Signed user key /home/user/.ssh/id_rsa-cert.pub: id "netconfusers" serial 0 for lab,netconf-
user valid from 2024-04-11T17:47:00 to 2025-04-16T17:48:20
```

4. Verify that the user certificate is created.

```
user@nms:~$ ssh-keygen -Lf path-to-user-cert
```

For the previous step, the command creates the **id_rsa-cert.pub** file.

```
lab@nms:~$ ssh-keygen -Lf /home/lab/.ssh/id_rsa-cert.pub
/home/lab/.ssh/id_rsa-cert.pub:
        Type: ssh-rsa-cert-v01@openssh.com user certificate
        Public key: RSA-CERT SHA256:SMInnJm8SXfsYLm7NvQZqcaGpuXc9QuraKZSsewdFKM
        Signing CA: RSA SHA256:0GbR/Lar1eH3/lULO66XSikD+gSwbibgTtFdnHNwlKg (using rsa-
sha2-512)
        Key ID: "netconfusers"
        Serial: 0
        Valid: from 2024-04-11T17:51:00 to 2025-04-16T17:52:45
        Principals:
                lab
                netconf-user
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

5. Copy the user certificate to the network management system, if the devices are different.

```
$ scp path-to-user-cert user@nms-ip:/home/user/.ssh
```

6. Ensure the user certificate file owner and permissions are correct.

```
user@nms:~$ sudo chown user:group /home/user/.ssh/id_rsa-cert.pub
user@nms:~$ sudo chmod 644 /home/user/.ssh/id_rsa-cert.pub
```

7. Add the SSH key and certificate identities to the ssh-agent application, if they are not added automatically. For example:

```
lab@nms:~$ ssh-add
Identity added: /home/lab/.ssh/id_rsa (lab@nms)
Certificate added: /home/lab/.ssh/id_rsa-cert.pub (netconfusers)
```

Generate the Host Certificate

The Junos device stores SSH keys in the **/etc/ssh** directory. To generate the host certificate, first copy the host's public key to the CA server, and then use the host CA private key to sign the host public key.

1. On the Junos device, verify the host public keys.

```
user@host> file list /etc/ssh/*.pub
/etc/ssh/ssh_host_ec_p384_key.pub
/etc/ssh/ssh_host_ec_p521_key.pub
/etc/ssh/ssh_host_ecdsa_key.pub
/etc/ssh/ssh_host_ed25519_key.pub
/etc/ssh/ssh_host_rsa_key.pub
```

2. Copy the Junos device's public key to the CA server that has the CA signing keys.

```
user@host> scp /etc/ssh/ssh_host_rsa_key.pub user@ca-server-ip:/var/tmp/
```

3. Generate the host certificate.

   Use the host CA private key to sign the host's public key, and specify the `-h` option to create a host certificate.

```
user@nms:~$ ssh-keygen -s path-to-host-CA-private-key -I key-identity -h path-to-host-public-
key
```

The following command uses the CA private key stored in **host_ca** to sign the host's public key stored in **ssh_host_rsa_key.pub**.

```
user@nms:~$ sudo ssh-keygen -s /root/.ssh/host_ca -I ssh_server -h /var/tmp/
ssh_host_rsa_key.pub
Signed host key /var/tmp/ssh_host_rsa_key-cert.pub: id "ssh_server" serial 0 valid forever
```

4. Verify that the host certificate is created.

```
user@nms:~$ ls path-to-host-certificate
```

For the previous step, the command creates the **ssh_host_rsa_key-cert.pub** file.

```
user@nms:~$ ls /var/tmp/ssh_host_rsa_key-cert.pub
/var/tmp/ssh_host_rsa_key-cert.pub
```

5. Copy the host certificate to the Junos device.

```
user@nms:~$ scp path-to-host-certificate user@device-ip:/var/tmp/
```

For example:

```
user@nms:~$ scp /var/tmp/ssh_host_rsa_key-cert.pub user@198.51.100.1:/var/tmp/
```

6. On the Junos device, configure the host certificate.

```
[edit system services ssh]
user@host# set host-certificate-file /var/tmp/ssh_host_rsa_key-cert.pub
user#host# commit
```

When you configure this statement, the device adds the host certificate to the **/var/etc/ssh_host_ca.pub** file.

Configure the Authorized Principals

User certificates can be tied to specific users, or principals. Junos devices require that the user certificate specify one or more authorized principals. You must also configure authorized principals on the Junos device. The Junos device authenticates the certificate if at least one of the authorized principals configured on the device matches an authorized principal in the certificate.

You can configure the list of authorized principals using several methods. You can load a file that contains the list of authorized principals, each on a separate line. You can also configure the authorized principal list directly in the configuration.

To configure the authorized principals:

1. (Optional) Create the authorized principals file and copy it to the Junos device.

   ```
   user@host> file show /var/tmp/auth_principals_list
   lab
   netconf-user
   ```

2. Configure the authorized principals that can access the system using SSH certificates.

   - To load a file containing the authorized principals, configure the `authorized-principals-file` statement and specify the path of the file to load.

     ```
     [edit system services ssh]
     user@router# set authorized-principals-file filepath
     ```

     For example:

     ```
     [edit system services ssh]
     user@router# set authorized-principals-file /var/tmp/auth_principals_list
     ```

   - To configure one or more authorized principals directly in the configuration, use the `authorized-principals` statement and specify a principal name or a list of principal names.

     ```
     [edit system services ssh]
     user@host# set authorized-principals principal
     user@host# set authorized-principals [principal1 principal2 principal3]
     ```

     For example:

     ```
     [edit system services ssh]
     user@host# set authorized-principals [lab netconf-user]
     ```

3. Commit the configuration.

   ```
   [edit]
   user@host# commit
   ```

   When you configure authorized principals, the device adds the list of authorized principals to the **/var/etc/ssh_authorized_principals** file.

Enable the NETCONF Service

To configure the NETCONF service for NETCONF Call Home on the Junos device:

1. Enable the NETCONF service on either the default NETCONF port (830) or a user-defined port.

   - To use the default NETCONF port (830), do not specify a port number.

     ```
     [edit system services]
     user@host# set netconf ssh
     ```

   - To use a specific port, configure the port number.

     The port can range from 1 through 65535, however, you should avoid configuring access on a port that is normally assigned for another service .

     ```
     [edit system services]
     user@host# set netconf ssh port port-number
     ```

2. Configure RFC-compliant NETCONF sessions.

   ```
   [edit system services]
   user@host# set netconf rfc-compliant
   ```

3. Configure YANG-compliant NETCONF sessions.

   ```
   [edit system services]
   user@host# set netconf yang-compliant
   ```

4. Commit the configuration:

```
[edit system services]
user@host# commit
```

Configure the Junos Device to Connect to the NETCONF Call Home Client

In a NETCONF call home scenario, the roles of the TCP server and client are reversed. You configure the Junos device to initiate, establish, and maintain a connection with a predefined set of network management systems. NETCONF Call Home uses outbound SSH to connect to the specified NETCONF clients.

Before you begin:

- If you use SSH certificates for authentication, configure the user CA certificates, the host certificate, and the authorized principals.

NOTE: The NETCONF call home client configuration registers the SSH certificate configuration at the time that you commit the client configuration. You must configure the host certificate and at least one user CA certificate and one authorized principal at the time that you commit the client configuration. If you add these initial values later, you must deactivate and commit the client configuration and then activate and commit the client configuration for the values to take effect. If the initial values are configured when you first activate the client configuration, then the client configuration will automatically register any updates to the existing values.

To configure a NETCONF Call Home client on the Junos device:

1. At the `[edit system services ssh]` hierarchy level, set the SSH version to v2.

```
[edit system services ssh]
user@host# set protocol-version v2
```

2. Navigate to the outbound SSH client hierarchy, and define an identifier that uniquely identifies the NETCONF Call Home client.

```
[edit]
user@host# edit system services outbound-ssh client client-id
```

For example:

```
[edit]
user@host# edit system services outbound-ssh client nms1
```

3. Configure the hostname or the IPv4 or IPv6 address for the NETCONF client. Additionally, specify the port on which the NETCONF client (acting as the TCP server) is listening for connection requests.

```
[edit system services outbound-ssh client nms1]
user@host# set address port port
```

For example:

```
[edit system services outbound-ssh client nms1]
user@host# set 198.51.100.10 port 4334
```

4. Define the device identifier, which is a user-defined string that identifies the Junos device to the client during the initiation sequence.

```
[edit system services outbound-ssh client nms1]
user@host# set device-id device-id
```

For example:

```
[edit system services outbound-ssh client nms1]
user@host# set device-id router1
```

5. Define the method used to reestablish a disconnected outbound SSH connection as `sticky` or `in-order`.

```
[edit system services outbound-ssh client nms1]
user@host# set reconnect-strategy in-order
```

6. Configure the NETCONF service for the session.

```
[edit system services outbound-ssh client nms1]
user@host# set services netconf
```

7. (Optional) Configure the Junos device to send keepalive messages to the management server.

   ```
   [edit system services outbound-ssh client nms1]
   user@host# set keep-alive retry num timeout seconds
   ```

   For example, the following configuration sends a keepalive signal every 15 seconds (the default), and terminates the SSH connection after sending 3 messages (the default) that do not receive a response.

   ```
   [edit system services outbound-ssh client nms1]
   user@host# set keep-alive timeout 15 retry 3
   ```

8. (Optional) Configure the `secret` statement if you want pass the device's public key to the management server during session initialization.

   ```
   [edit system services outbound-ssh client nms1]
   user@host# set secret password
   ```

   When you configure this statement, the device passes its public key along with a SHA1 hash derived in part from the `secret` statement to the management server during session initialization. The client application can use the shared secret and hash to verify whether the presented host key is from the device identified by the `device-id` statement.

9. Repeat the preceding steps for each NETCONF Call Home client that will manage the Junos device.
10. Commit the configuration.

   ```
   [edit system services outbound-ssh client nms1]
   user@R0# commit and-quit
   ```

After you configure the device to connect to the NETCONF Call Home client and commit the configuration, the device attempts to initiate a TCP/IP connection with the client. The device continues to create this connection until successful or until the NETCONF call home client configuration is deleted or deactivated.

Each time the Junos device establishes an outbound SSH connection, it sends an initiation sequence that identifies the device to the client. Within this transmission is the value of `device-id`. When you do not configure the `secret` statement, the initiation sequence is:

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: <device-id>\r\n
```

If you configure the `secret` statement, the initiation sequence includes the Junos device's public key, and a SHA1 hash derived in part from the `secret` statement. The client can compute the expected hash value and compare it to the HMAC value to verify that the presented host key is from the device identified by the `device-id` statement.

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: <device-id>\r\n
HOST-KEY: <public-host-key>\r\n
HMAC:<HMAC(pub-SSH-host-key, <secret>)>\r\n
```

The client authenticates the Junos device by validating the presented host key or certificate. Similarly, the NETCONF server authenticates the client. The server first checks certificate-based authentication, then key-based authentication, and finally password-based authentication. After the SSH session is established, the client starts the NETCONF session.

NETCONF Sessions

SUMMARY

Understand NETCONF sessions on Junos devices.

IN THIS SECTION

You can use the Network Configuration Protocol (NETCONF) to manage network devices. The following sections provide an overview of NETCONF sessions on Junos devices.

Connect to the NETCONF Server Using SSH

IN THIS SECTION

The most common method for connecting to the NETCONF server is to use SSH. Before a NETCONF client can connect to the NETCONF server using SSH, you must satisfy the requirements described in "Establish an SSH Connection for a NETCONF Session" on page 41. When the prerequisites are satisfied, a NETCONF client can connect to the NETCONF server using one of the following methods:

SSH Library Routines

A NETCONF client uses SSH library routines to establish an SSH connection to the NETCONF server, provide authentication, and create a channel that acts as an SSH subsystem for the NETCONF session. Providing instructions for using library routines is beyond the scope of this document.

ssh Command

You can establish a NETCONF session as an SSH subsystem with a dedicated port. Alternatively, you can establish a NETCONF session over the default SSH port and use pseudo-tty allocation. Using an SSH subsystem over a dedicated port enables the device to easily identify and filter NETCONF traffic.

However, using the default SSH port with pseudo-tty allocation can provide visibility to the session, for example, when issuing the `show system users` operational command.

The application must include code to intercept the NETCONF server's prompt for the password or passphrase. For example, the application can use a utility such as the `expect` command.

- To establish a NETCONF session as an SSH subsystem over the default NETCONF port (830), the client application issues the following command:

  ```
  ssh user@hostname -p 830 -s netconf
  ```

  The `-p` option defines the port number on which the NETCONF server listens. This option can be omitted if you enabled access to SSH over the default port.

  The `-s` option establishes the NETCONF session as an SSH subsystem.

- To establish a NETCONF session over the default SSH port (22) and use pseudo-tty allocation, the client application issues the following command:

  ```
  ssh user@hostname -t netconf
  ```

  NOTE: Using multiple `-t` options forces pseudo-tty allocation even if SSH has no local tty.

Start a NETCONF Session

IN THIS SECTION

Each NETCONF session begins with a handshake in which the NETCONF server and the client application specify their supported NETCONF capabilities. The following sections describe how to start a NETCONF session.

Exchanging <hello> Tag Elements

The NETCONF server and client application each begin by emitting a `<hello>` tag element to specify which operations, or *capabilities*, they support from among those defined in the NETCONF specification. The client application must emit the `<hello>` element before any other element and must only emit it once.

The `<hello>` tag encloses the following elements:

- `<capabilities>`—List of `<capability>` elements, which each define a supported function.
- `<session-id>`—UNIX process ID (PID) of the NETCONF server for the session.

Each capability defined in the NETCONF specification is represented in a `<capability>` element by a uniform resource name (URN). Capabilities defined by individual vendors are represented by uniform resource identifiers (URIs), which can be URNs or URLs. The NETCONF server emits a `<hello>` element similar to the following output:

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>
       urn:ietf:params:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>
       urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file
    </capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>22062</session-id>
</hello>
```

The URIs in the `<hello>` element indicate the supported capabilities. Table 8 on page 120 lists some common capabilities.

Table 8: Common Capabilities

Capability	Description
`urn:ietf:params:netconf:base:1.0`	The NETCONF server supports the basic operations and elements defined in the base NETCONF specification.
`urn:ietf:params:netconf:base:1.1`	The NETCONF session is compliant with RFC 6242, which supports the chunked framing mechanism for message framing.
`urn:ietf:params:netconf:capability:candidate:1.0`	The NETCONF server supports operations on a candidate configuration.
`urn:ietf:params:netconf:capability:confirmed-commit:1.0`	The NETCONF server supports confirmed commit operations. For more information, see "Commit the Candidate Configuration Only After Confirmation Using NETCONF" on page 325
`urn:ietf:params:netconf:capability:validate:1.0`	The NETCONF server supports the validation operation, which verifies the syntactic correctness of a configuration without actually committing it. For more information, see "Verify the Candidate Configuration Syntax Using NETCONF" on page 322.
`urn:ietf:params:netconf:capability:url:1.0?protocol=http,ftp,file`	The NETCONF server accepts configuration data stored in a file. It can retrieve files both from its local file system and from remote machines by using HTTP or FTP. For more information, see "Upload and Format Configuration Data in a NETCONF Session" on page 279.

Table 8: Common Capabilities *(Continued)*

Capability	Description
`http://xml.juniper.net/netconf/junos/1.0`	The NETCONF server supports: • Operations defined in the Junos XML API for requesting and changing operational information • Junos XML protocol operations for requesting or changing configuration information NETCONF client applications should use only native NETCONF operations and supported Junos XML protocol extensions for configuration functions. The semantics of corresponding Junos XML protocol operations and NETCONF protocol operations are not necessarily identical, so using Junos XML protocol configuration operations other than the documented supported extensions can lead to unexpected results.
`http://xml.juniper.net/dmi/system/1.0`	The NETCONF server supports the operations defined in the Device Management Interface (DMI) specification.

By default, the NETCONF server does not advertise supported YANG modules in the NETCONF capabilities exchange. To advertise supported YANG modules, configure one or more of the following statements at the `[edit system services netconf hello-message yang-module-capabilities]` hierarchy level:

- `advertise-custom-yang-modules`—Advertise third-party YANG modules installed on the device.
- `advertise-native-yang-modules`—Advertise Junos OS native YANG modules.
- `advertise-standard-yang-modules`—Advertise standard YANG modules supported by the device, for example, OpenConfig modules.

To comply with the NETCONF specification, the client application also emits a `<hello>` element to define the capabilities it supports. It does not include the `<session-id>` element:

```
<hello>
<capabilities>
    <capability>first-capability</capability>
    <!-- tag elements for additional capabilities -->
</capabilities>
```

```
</hello>
]]>]]>
```

NETCONF sessions use a framing mechanism to separate the messages that the NETCONF server and client send within the session. By default, a NETCONF session with a Junos device uses the character sequence]]>]]> as a message separator. However, if you configure RFC 6242-compliant NETCONF sessions, and both peers advertise the `:base:1.1` capability in the capabilities exchange, the NETCONF session uses chunked framing for the remainder of the session. Chunked framing is a standardized framing mechanism that ensures that character sequences within XML elements are not misinterpreted as message boundaries. For more information, see "Configure RFC-Compliant NETCONF Sessions" on page 143.

The session continues when the client application sends a request to the NETCONF server. The NETCONF server does not emit any elements after session initialization except in response to the client application's requests.

Verifying Compatibility

Exchanging `<hello>` tag elements enables the NETCONF server and client to determine if they support the same capabilities. In addition, we recommend that the client application determine the Junos OS version running on the NETCONF server. After emitting its `<hello>` tag, the client application can emit the `<get-software-information>` request.

```
<rpc>
    <get-software-information/>
</rpc>
]]>]]>
```

The NETCONF server returns the `<software-information>` element, which encloses different tags depending on the Junos OS variant. Junos OS returns the `<host-name>`, `<product-name>`, and `<junos-version>` elements. The server also returns a `<package-information>` element for each software module. The `<comment>` elements specify the Junos OS release and the build date. In the following example, the release is `24.4R1.9`.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           xmlns:junos="http://xml.juniper.net/junos/24.4R1.9/junos">
<software-information>
  <host-name>balerion-mx204-a</host-name>
  <product-model>mx204</product-model>
  <product-name>JNP204 [MX204]</product-name>
  <os-name>junos</os-name>
  <junos-version>24.4R1.9</junos-version>
  <package-information>
    <name>os-kernel</name>
    <package-name>os-kernel-prd-x86-64-20241104.1ed86e6_builder_bsd15_244</package-name>
    <comment>JUNOS OS Kernel 64-bit  [20241104.1ed86e6_builder_bsd15_244]</comment>
  </package-information>
  <package-information>
    <name>junos-modules</name>
    <package-name>junos-modules-x86-64-20241219.060016_builder_junos_244_r1</package-name>
    <comment>JUNOS modules [20241219.060016_builder_junos_244_r1]</comment>
  </package-information>
  <!-- <package-information> tag elements for additional modules -->
</software-information>
</rpc-reply>
]]>]]>
```

Normally, the version is the same for all Junos OS modules running on the device (we recommend this configuration for predictable routing performance). Therefore, verifying the version number of just one module is usually sufficient.

The client application is responsible for determining how to handle any differences in version or capabilities. For fully automated performance, include code in the client application that determines whether it supports the same capabilities and Junos OS version as the NETCONF server. When there are differences, determine which of the following options is appropriate, and implement the corresponding response:

- **Ignore differences**—Ignore differences in capabilities and Junos OS version, and do not alter the client application's behavior to accommodate the NETCONF server. A difference in Junos OS versions does not necessarily make the server and client incompatible, so this is often a valid approach. Similarly, it is a valid approach if the capabilities that the client application does not support are operations that are always initiated by a client, such as validation of a configuration and confirmed commit. In that case, the client maintains compatibility by not initiating the operation.

- **Alter standard behavior to be compatible with the NETCONF server**—If the client application is running a later version of the Junos OS, for example, it can choose to emit only NETCONF and Junos XML tag elements that represent the software features available in the NETCONF server's version of Junos OS.

- **End the NETCONF session and terminate the connection**—Use this option if you decide that it is not practical to accommodate the NETCONF server's version or capabilities.

Send Requests to the NETCONF Server

IN THIS SECTION

How to Send Requests

To initiate a request to the NETCONF server, a client application emits the following:

- Opening `<rpc>` tag
- One or more tag elements that represent the particular request
- Closing `</rpc>` tag

```
<rpc>
    <!-- tag elements representing a request -->
</rpc>
]]>]]>
```

The application encloses each request in its own separate pair of opening `<rpc>` and closing `</rpc>` tags. Each request must constitute a well-formed XML document by including only compliant and correctly ordered tag elements. The NETCONF server ignores any newline characters, spaces, or other white space characters that occur between tag elements in the tag stream, but it preserves white space within tag elements.

Optionally, a client application can include one or more attributes of the form *attribute-name*="*value*" in the opening `<rpc>` tag for each request. The NETCONF server echoes each attribute, unchanged, in the opening `<rpc-reply>` tag in which it encloses its response.

A client application can use this feature to associate requests and responses by including an attribute in each opening `<rpc>` request tag that assigns a unique identifier. The NETCONF server echoes the attribute in its opening `<rpc-reply>` tag, making it easy to map the response to the initiating request. The NETCONF specification specifies the name `message-id` for this attribute.

Although operational and configuration requests conceptually belong to separate classes, a NETCONF session does not have distinct modes that correspond to CLI operational and configuration modes. Each request tag element is enclosed within its own `<rpc>` tag, so a client application can freely alternate operational and configuration requests. A client application can make three classes of requests: operational requests, configuration information requests, and configuration change requests.

Operational Requests

Operational requests are requests for information about the status of a device. Operational requests correspond to the Junos OS CLI operational mode commands. The Junos XML API defines a request tag element for many CLI commands. For example, the `<get-interface-information>` tag element corresponds to the `show interfaces` command, and the `<get-chassis-inventory>` tag element requests the same information as the `show chassis hardware` command.

The following RPC requests detailed information about interface ge-2/3/0:

```
<rpc>
    <get-interface-information>
        <interface-name>ge-2/3/0</interface-name>
        <detail/>
    </get-interface-information>
</rpc>
]]>]]>
```

For more information about operational requests, see "Request Operational Information Using NETCONF" on page 386.

For information about Junos XML request tags, see the XML API Explorer.

Configuration Information Requests

Configuration information requests are requests for information about the device's candidate configuration, a private configuration, the ephemeral configuration, or the committed (active) configuration. The candidate and committed configurations diverge when there are uncommitted changes to the candidate configuration.

The NETCONF protocol defines the `<get-config>` operation for retrieving configuration information. The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy.

The following example requests information from the `[edit system login]` hierarchy level of the candidate configuration:

```
<rpc>
    <get-config>
        <source>
            <candidate/>
        </source>
        <filter type="subtree">
            <configuration>
                <system>
                    <login/>
                </system>
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
```

For more information about configuration information requests, see "Request Configuration Data Using NETCONF" on page 400.

For a summary of the available configuration tag elements, see the XML API Explorer.

Configuration Change Requests

Configuration change requests are requests to change the configuration, or to commit those changes on the device. The NETCONF protocol defines the `<edit-config>` and `<copy-config>` operations for changing configuration information. The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy.

The following example creates a Junos OS user account called `admin` at the `[edit system login]` hierarchy level in the candidate configuration:

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <config>
            <configuration>
                <system>
```

```
                      <login>
                          <user>
                              <name>admin</name>
                              <full-name>Administrator</full-name>
                              <class>superuser</class>
                          </user>
                      </login>
                      <login/>
                  </system>
              </configuration>
          </config>
      </edit-config>
  </rpc>
  ]]>]]>
```

For more information about configuration change requests, see "Edit the Configuration Using NETCONF" on page 277.

For a summary of the available configuration tag elements, see the XML API Explorer.

Parse the NETCONF Server Response

IN THIS SECTION

NETCONF Server Response Overview

In a NETCONF session, a client application sends RPCs to the NETCONF server to request information and manage the device configuration. The NETCONF server encloses its response to each client request in a separate pair of opening `<rpc-reply>` and closing `</rpc-reply>` tags. Each response constitutes a well-formed XML document.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" \
                xmlns:junos="http://xml.juniper.net/junos/release/junos" \
                [echoed attributes]>
```

```
    <!-- tag elements representing a response -->
</rpc-reply>
]]>]]>
```

The `xmlns` attribute in the opening `<rpc-reply>` tag defines the namespace for enclosed tag elements that do not have the `junos:` prefix in their names and that are not enclosed in a child container tag that has the `xmlns` attribute with a different value.

NOTE: If you configure the `rfc-compliant` statement on the device, the NETCONF server explicitly declares the NETCONF namespace, which is bound to the `nc` prefix, and qualifies all NETCONF tags in its replies with the prefix.

The `xmlns:junos` attribute defines the default namespace for enclosed Junos XML tag elements that are qualified by the `junos:` prefix. The *release* variable in the URI represents the Junos OS release that is running on the NETCONF server device, for example 20.4R1.

Client applications must include code for parsing the stream of response tag elements coming from the NETCONF server, either processing the elements as they arrive or storing them until the response is complete. The NETCONF server returns three classes of responses: operational responses, configuration information responses, and configuration change responses.

Operational Responses

Operational responses are responses to requests for information about the status of a device. They correspond to the output from CLI operational commands.

The Junos XML API defines response tag elements for all defined operational request tag elements. For example, the NETCONF server returns the information requested by the `<get-interface-information>` tag in a response tag called `<interface-information>`. Similarly, the server returns the information requested by the `<get-chassis-inventory>` tag in a response tag called `<chassis-inventory>`.

By default, the server returns operational responses in XML format. The client application can also request responses in formatted ASCII or in JSON. For more information about formatting operational responses, see "Specify the Output Format for Operational Information Requests in a NETCONF Session" on page 390.

The following sample operational response includes information about the interface ge-2/3/0:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
                xmlns:junos="http://xml.juniper.net/junos/20.4R1/junos">
    <interface-information \
                xmlns="http://xml.juniper.net/junos/20.4R1/junos-interface">
        <physical-interface>
```

```
            <name>ge-2/3/0</name>
            <!-- other data tag elements for the ge-2/3/0 interface -  ->
        </physical-interface>
    </interface-information>
</rpc-reply>
]]>]]>
```

For more information about the `xmlns` attribute and the contents of operational response tag elements, see "Request Operational Information Using NETCONF" on page 386.

Configuration Information Responses

Configuration information responses are responses to requests for information about the device's current configuration. The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy.

The following sample response includes configuration data at the `[edit system login]` hierarchy level:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
    xmlns:junos="http://xml.juniper.net/junos/20.4R1/junos">
    <data>
        <configuration attributes>
            <system>
                <login>
                    <user>
                        <name>admin</name>
                        <full-name>Administrator</full-name>
                        <!-- other data tag elements for the admin user -->
                    </user>
                </login>
            </system>
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

Configuration Change Responses

Configuration change responses are responses to requests that change the state or contents of the device configuration. The NETCONF server indicates successful execution of a request by returning the `<ok/>` tag within the `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the operation fails, the `<rpc-reply>` tag element instead encloses an `<rpc-error>` element that describes the cause of the failure.

Parse Response Tag Elements Using a Standard API in NETCONF and Junos XML Protocol Sessions

In a NETCONF or Junos XML protocol session, client applications can handle incoming XML tag elements by feeding them to a parser that is based on a standard API such as the Document Object Model (DOM) or Simple API for XML (SAX). Describing how to implement and use a parser is beyond the scope of this documentation.

Routines in the DOM accept incoming XML and build a tag hierarchy in the client application's memory. There are also DOM routines for manipulating an existing hierarchy. DOM implementations are available for several programming languages, including C, C++, Perl, and Java. For detailed information, see the *Document Object Model (DOM) Level 1 Specification* from the World Wide Web Consortium (W3C) at http://www.w3.org/TR/REC-DOM-Level-1/ . Additional information is available from the Comprehensive Perl Archive Network (CPAN) at https://metacpan.org/search?q=dist:XML-DOM+dom.

One potential drawback with DOM is that it always builds a hierarchy of tag elements, which can become very large. If a client application needs to handle only one subhierarchy at a time, it can use a parser that implements SAX instead. SAX accepts XML and feeds the tag elements directly to the client application, which must build its own tag hierarchy. For more information, see the official SAX website at http://sax.sourceforge.net/ .

Handle an Error or Warning in a NETCONF Session

A client application sends RPCs to the NETCONF server to request information and manage the device configuration. The NETCONF server sends a response for each client request. If the server encounters an error condition, it emits an `<rpc-error>` element containing child elements that describe the error.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rpc-error>
```

```
        <error-severity>error-severity</error-severity>
        <error-path>error-path</error-path>
        <error-message>error-message</error-message>
        <error-info>
            <bad-element>command-or-statement</bad-element>
        </error-info>
    <rpc-error>
</rpc-reply>
]]>]]>
```

The `<rpc-error>` element can include the following child elements:

- `<bad-element>` identifies the command or configuration statement that was being processed when the error or warning occurred. For a configuration statement, `<error-path>` specifies the statement's parent hierarchy level.
- `<error-message>` describes the error or warning in a natural-language text string.
- `<error-path>` specifies the path to the Junos OS configuration hierarchy level at which the error or warning occurred.
- `<error-severity>` indicates the severity of the event that caused the NETCONF server to return the `<rpc-error>` tag element. The two possible values are `error` and `warning`.

An error can occur while the server is performing any of the following operations. The server can send a different combination of child tag elements in each case.

- Processing an operational request submitted by a client application
- Opening, locking, changing, committing, or closing a configuration as requested by a client application
- Parsing configuration data submitted by a client application in an `<edit-config>` tag element

Client applications must be prepared to receive and handle an `<rpc-error>` element at any time. The information in any response tag elements already received and related to the current request might be incomplete. The client application can include logic for deciding whether to discard or retain the information.

When the `<error-severity>` element has the value `error`, the usual response is for the client application to discard the information and terminate. When the `<error-severity>` tag element has the value `warning`, indicating that the problem is less serious, the usual response is for the client application to log the warning or pass it to the user and to continue parsing the server's response.

NOTE: When you configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level to enforce certain behaviors by the NETCONF server, the NETCONF server cannot return an RPC reply that includes both an `<rpc-error>` element and an `<ok/>` element. If the operation is successful, but the server reply would include one or more `<rpc-error>` elements with a severity level of warning in addition to the `<ok/>` element, then the warnings are omitted.

SEE ALSO

<rpc-error> | 214

Lock and Unlock the Candidate Configuration

IN THIS SECTION

When a client application is requesting or changing configuration information, it can use one of the following methods to access the candidate configuration:

- Lock the candidate configuration, which prevents other users or applications from changing the shared configuration database until the application releases the lock. This is equivalent to the CLI `configure exclusive` command.
- Change the candidate configuration without locking it. We do not recommend this method, because of the potential for conflicts with changes made by other applications or users that are editing the shared configuration database at the same time.

If an application is simply requesting configuration information and not changing it, locking the configuration is not required. The application can begin requesting information immediately. However, if it is important that the information being returned not change during the session, it is appropriate to lock the configuration.

Locking the Candidate Configuration

Locking the candidate configuration prevents other users or applications from changing the candidate configuration until the lock is released. We recommend that you lock the configuration before making changes, particularly on devices where multiple users are authorized to change the configuration. A commit operation applies to all changes in the candidate configuration, not just those made by the user or application that requests the commit. Allowing multiple users or applications to make changes simultaneously can lead to unexpected results.

To lock the candidate configuration, a client application executes the `<lock>` operation with `<target>` set to `<candidate/>` as follows:

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
]]>]]>
```

The RPC is equivalent to the CLI `configure exclusive` command.

The NETCONF server confirms that it has locked the candidate by returning the `<ok/>` tag.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot lock the configuration, the `<rpc-reply>` instead encloses an `<rpc-error>` element explaining the reason for the failure. Reasons for the failure can include the following:

- Another user or application has already locked the candidate configuration. The error message reports the NETCONF session identifier of the user or application.
- The candidate configuration already includes changes that have not yet been committed.

Only one application can hold the lock on the candidate configuration at a time. Other users and applications can read the candidate configuration while it is locked. The lock persists until either the client application unlocks the configuration by emitting the `<unlock>` tag element or the NETCONF session ends.

If the client application unlocks the candidate configuration before committing the changes, or if the NETCONF session ends for any reason before the changes are committed, the changes are automatically discarded. The candidate and committed configurations remain unchanged.

Unlocking the Candidate Configuration

As long as a client application holds a lock on the candidate configuration, other applications and users cannot change the candidate. To unlock the candidate configuration, the client application executes the `<unlock>` operation.

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

The NETCONF server confirms that it has unlocked the candidate by returning the `<ok/>` tag.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot unlock the configuration, the `<rpc-reply>` instead encloses an `<rpc-error>` element explaining the reason for the failure.

Terminate a NETCONF Session

In a NETCONF session, a client application's attempt to lock the candidate configuration can fail because another user or application already holds the lock. In this case, the NETCONF server returns an error message that includes the username and process ID (PID) for the entity that holds the existing lock.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rpc-error>
        <error-severity>error</error-severity>
        <error-message>
            configuration database locked by:
```

```
user terminal (pid PID) on since YYYY-MM-DD hh:mm:ss TZ, idle hh:mm:ss
            exclusive
        </error-message>
    </rpc-error>
</rpc-reply>
]]>]]>
```

If the client application has the Junos OS `maintenance` permission, it can end the session that holds the lock by executing the `<kill-session>` operation. The `<session-id>` element specifies the PID obtained from the error message.

```
<rpc>
    <kill-session>
        <session-id>PID</session-id>
    </kill-session>
</rpc>
]]>]]>
```

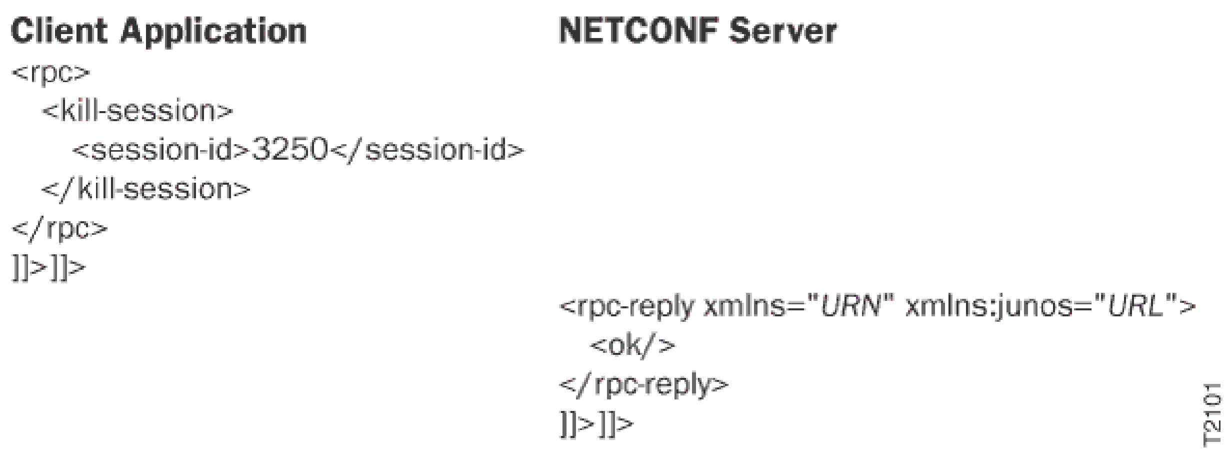
The NETCONF server confirms that it terminated the other session by returning the `<ok/>` tag.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

We recommend that the application include logic for determining whether it is appropriate to terminate another session. The logic might include factors such as the identity of the user or application that holds the lock or the length of idle time.

When a session is terminated, the NETCONF server that is servicing the session rolls back all uncommitted changes that have been made during the session. If a confirmed commit is pending (changes have been committed but not yet confirmed), the NETCONF server restores the configuration to its state before the confirmed commit instruction was issued.

The following example shows how to terminate another session:



```
<rpc>
   <kill-session>
      <session-id>3250</session-id>
   </kill-session>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
   <ok/>
</rpc-reply>
]]>]]>
```

End a NETCONF Session and Close the Connection

When a client application is finished making requests, it ends the NETCONF session by emitting the empty `<close-session/>` tag within an `<rpc>` element.

```
<rpc>
    <close-session/>
</rpc>
]]>]]>
```

The NETCONF server emits an `<rpc-reply>` element and the `<ok/>` tag.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

Because the connection to the NETCONF server is an SSH subsystem, it closes automatically when the NETCONF session ends.

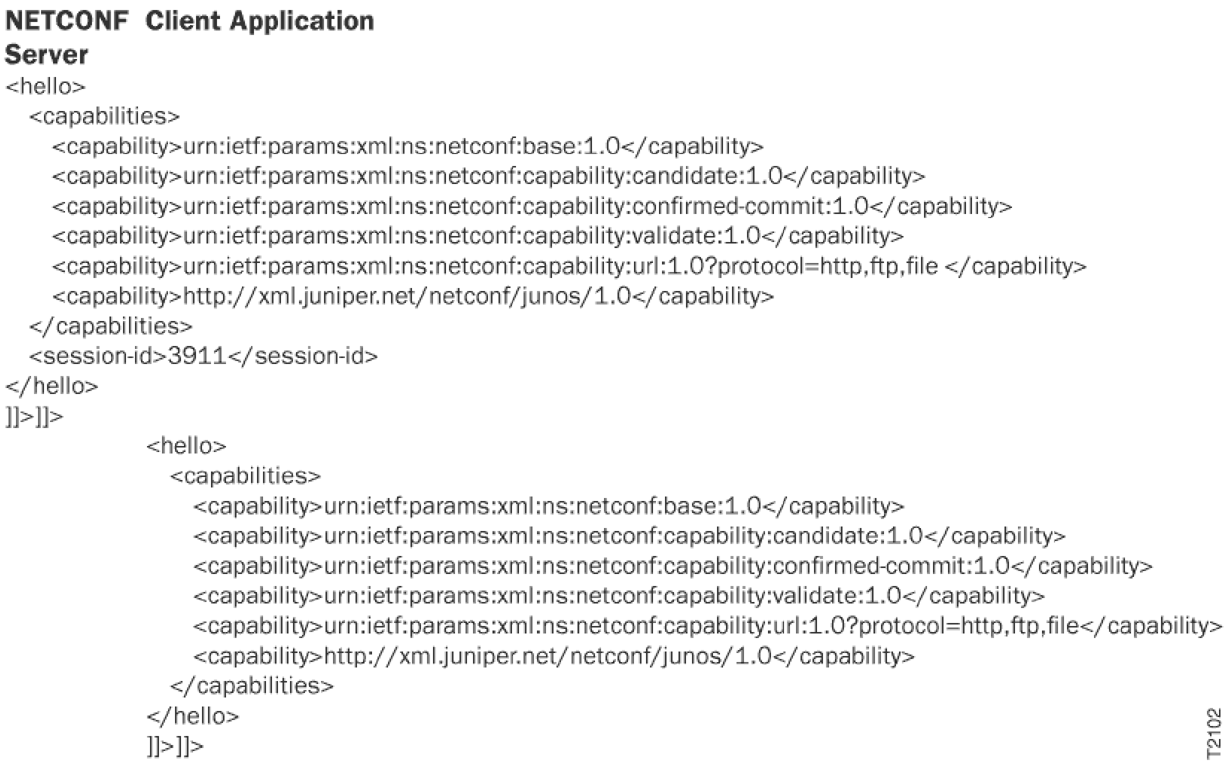
Sample NETCONF Session

IN THIS SECTION

The following sections describe the sequence of tag elements in a sample NETCONF session with a device running Junos OS. The client application begins by establishing a connection to a NETCONF server.

Exchanging Initialization Tag Elements

After the client application establishes a connection to a NETCONF server, the two exchange `<hello>` tag elements, as shown in the following example. For legibility, the example places the client application's `<hello>` tag element below the NETCONF server's. The two parties can actually emit their `<hello>` tag elements at the same time. For information about the `]]>]]>` character sequence used in this and the following examples, see "Generate Well-Formed XML Documents" on page 37. For a detailed discussion of the `<hello>` tag element, see "Start a NETCONF Session" on page 118.

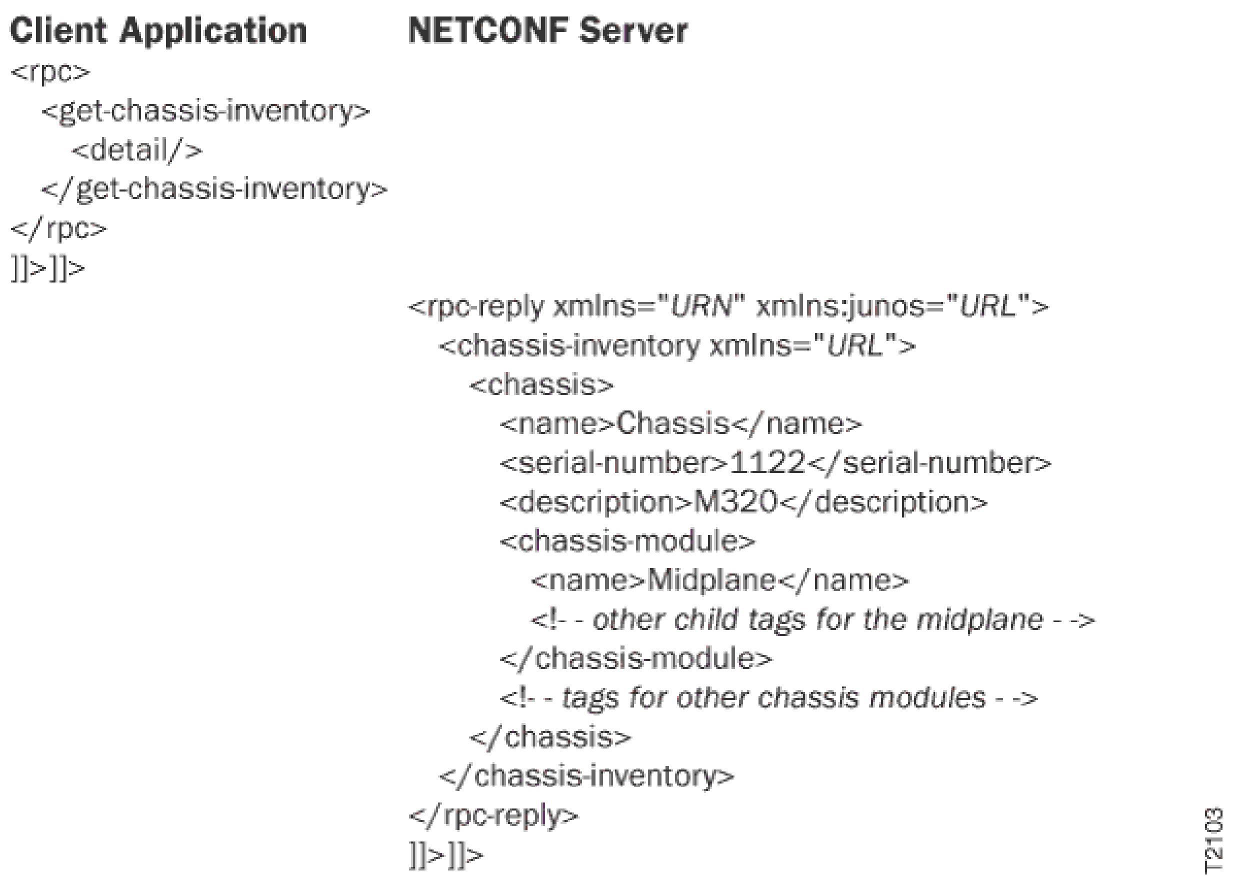


NETCONF Server   Client Application

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
  </capabilities>
  <session-id>3911</session-id>
</hello>
]]>]]>
            <hello>
              <capabilities>
                <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
                <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
                <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
                <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
                <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file</capability>
                <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
              </capabilities>
            </hello>
            ]]>]]>
```

Sending an Operational Request

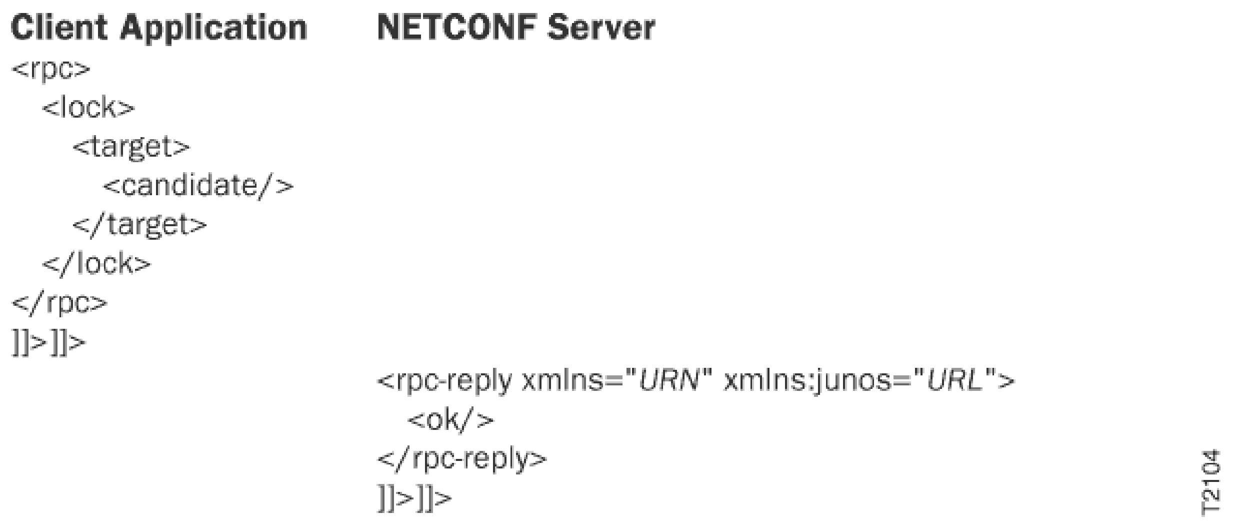
The client application emits the `<get-chassis-inventory>` tag element to request information about the device's chassis hardware. The NETCONF server returns the requested information in the `<chassis-inventory>` tag element.



```
Client Application          NETCONF Server
<rpc>
   <get-chassis-inventory>
      <detail/>
   </get-chassis-inventory>
</rpc>
]]>]]>
                            <rpc-reply xmlns="URN" xmlns:junos="URL">
                               <chassis-inventory xmlns="URL">
                                  <chassis>
                                     <name>Chassis</name>
                                     <serial-number>1122</serial-number>
                                     <description>M320</description>
                                     <chassis-module>
                                        <name>Midplane</name>
                                        <!- - other child tags for the midplane - ->
                                     </chassis-module>
                                     <!- - tags for other chassis modules - ->
                                  </chassis>
                               </chassis-inventory>
                            </rpc-reply>
                            ]]>]]>
```

Locking the Configuration

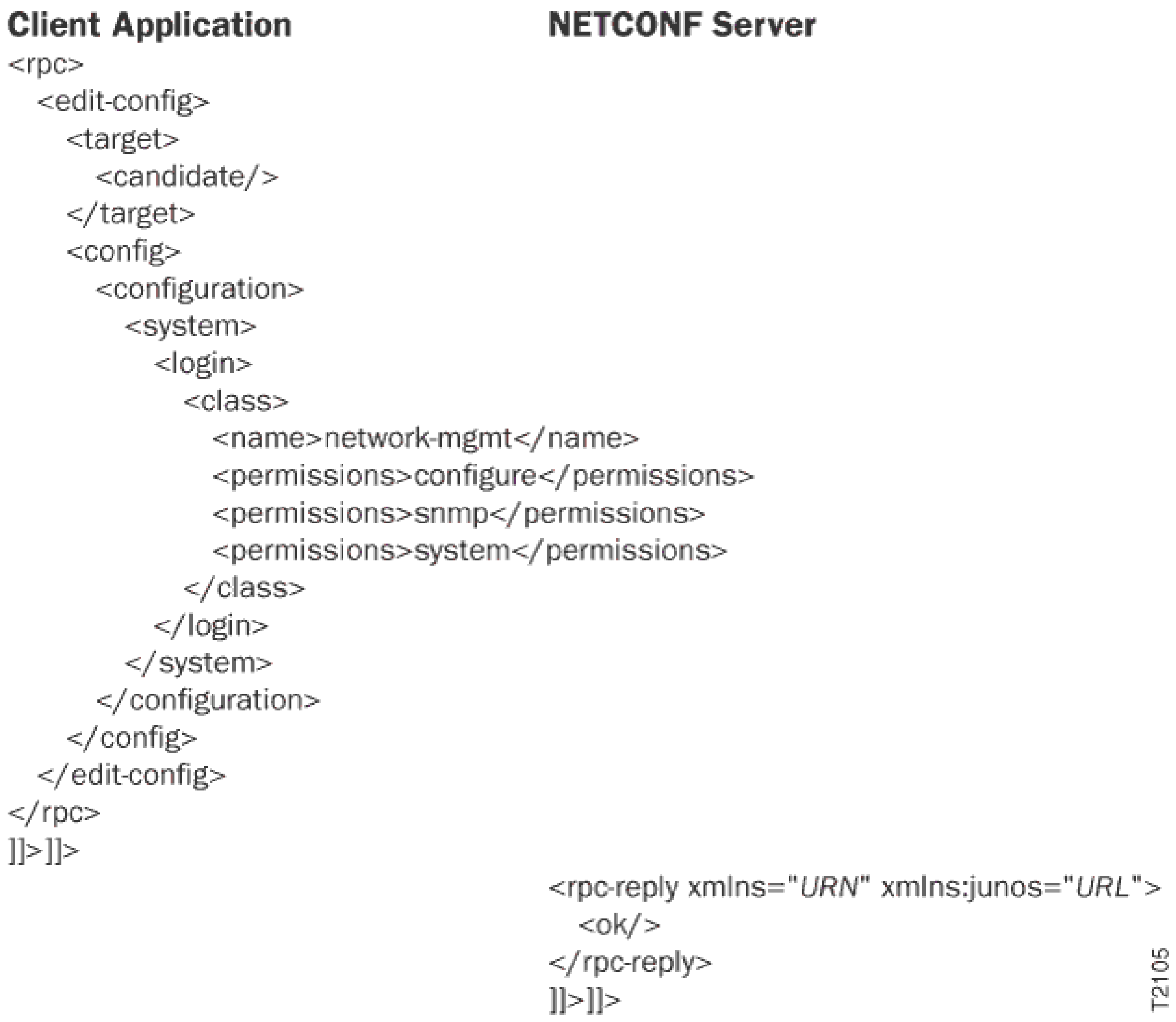
The client application then prepares to incorporate a change into the candidate configuration by emitting the `<lock/>` tag to prevent any other users or applications from altering the candidate configuration at the same time. To confirm that the candidate configuration is locked, the NETCONF server returns an `<ok/>` tag in an `<rpc-reply>` tag element.



```
Client Application       NETCONF Server
<rpc>
   <lock>
      <target>
         <candidate/>
      </target>
   </lock>
</rpc>
]]>]]>
                         <rpc-reply xmlns="URN" xmlns:junos="URL">
                            <ok/>
                         </rpc-reply>
                         ]]>]]>
```

Changing the Configuration

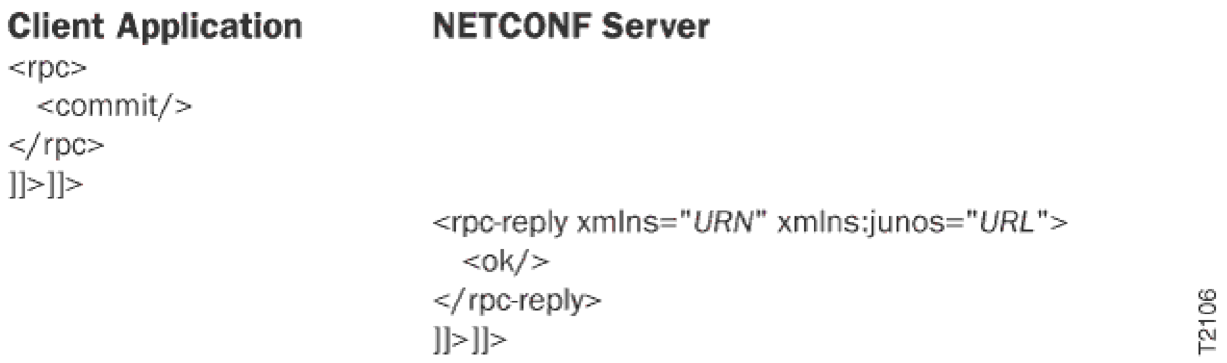
The client application now emits tag elements to create a new Junos OS login class called `network-mgmt` at the `[edit system login class]` hierarchy level in the candidate configuration. To confirm that the load operation was successful, the NETCONF server returns an `<ok/>` tag in an `<rpc-reply>` tag element.



```
Client Application                         NETCONF Server
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <login>
            <class>
              <name>network-mgmt</name>
              <permissions>configure</permissions>
              <permissions>snmp</permissions>
              <permissions>system</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
                                           <rpc-reply xmlns="URN" xmlns:junos="URL">
                                             <ok/>
                                           </rpc-reply>
                                           ]]>]]>
```

Committing the Configuration

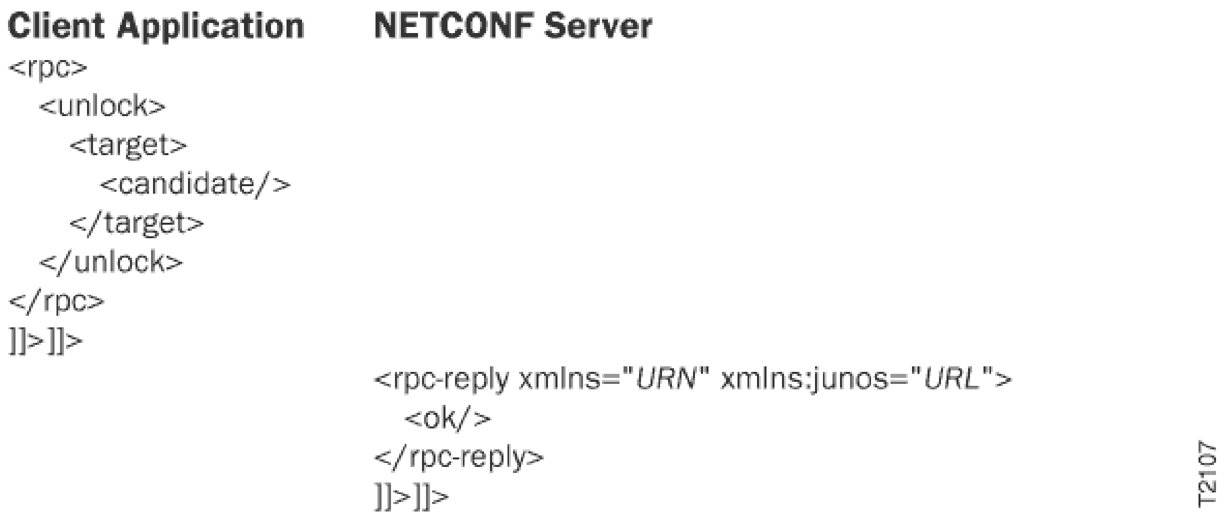
The client application then commits the candidate configuration. To confirm that the commit operation was successful, the NETCONF server returns an `<ok/>` tag in an `<rpc-reply>` tag element.



```
Client Application          NETCONF Server
<rpc>
  <commit/>
</rpc>
]]>]]>
                            <rpc-reply xmlns="URN" xmlns:junos="URL">
                              <ok/>
                            </rpc-reply>
                            ]]>]]>
```

Unlocking the Configuration

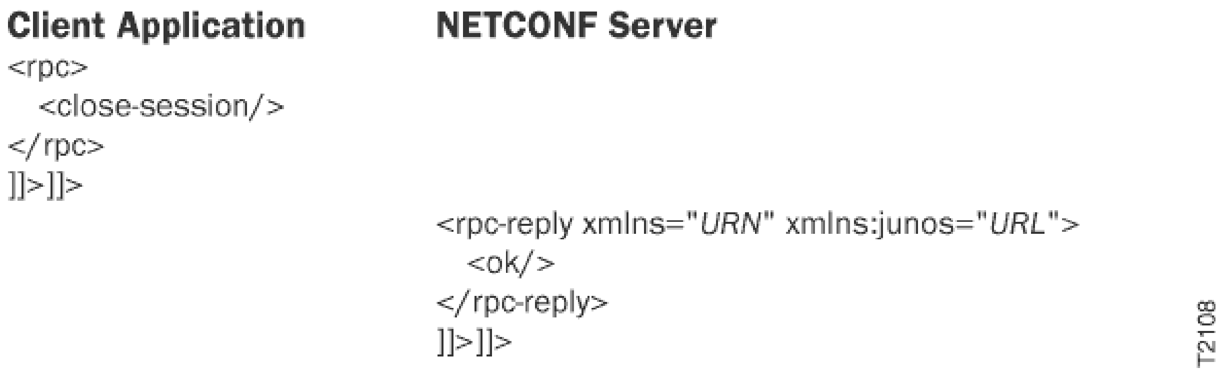
The client application unlocks (and by implication closes) the candidate configuration. To confirm that the unlock operation was successful, the NETCONF server returns an `<ok/>` tag in an `<rpc-reply>` tag element.



```
Client Application     NETCONF Server
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
                       <rpc-reply xmlns="URN" xmlns:junos="URL">
                         <ok/>
                       </rpc-reply>
                       ]]>]]>
```

Closing the NETCONF Session

The client application closes the NETCONF session by emitting the `<close-session>` tag.



```
Client Application          NETCONF Server
<rpc>
  <close-session/>
</rpc>
]]>]]>
                            <rpc-reply xmlns="URN" xmlns:junos="URL">
                              <ok/>
                            </rpc-reply>
                            ]]>]]>
```

RELATED DOCUMENTATION

NETCONF Sessions | 116

Generate Well-Formed XML Documents | 37

How Character Encoding Works on Juniper Networks Devices

Junos OS configuration data and operational command output might contain non-ASCII characters, which are outside of the 7-bit ASCII character set. When displaying operational or configuration data in certain formats or within a certain type of session, the software escapes and encodes these characters. The software escapes or encodes the characters using the equivalent UTF-8 decimal character reference.

The CLI attempts to display any non-ASCII characters in configuration data that is produced in text, set, or JSON format. The CLI also attempts to display these characters in command output that is produced in text format. In the exception cases, the CLI displays the UTF-8 decimal character reference instead. (Exception cases include configuration data in XML format and command output in XML or JSON format,) In NETCONF and Junos XML protocol sessions, you see a similar result if you request configuration data or command output that contains non-ASCII characters. In this case, the server returns the equivalent UTF-8 decimal character reference for those characters for all formats.

For example, suppose the following user account, which contains the Latin small letter n with a tilde (ñ), is configured on the device.

```
[edit]
user@host# set system login user mariap class super-user uid 2007 full-name "Maria Peña"
```

When you display the resulting configuration in text format, the CLI prints the corresponding character.

```
[edit]
user@host# show system login user mariap
full-name "Maria Peña";
uid 2007;
class super-user;
```

When you display the resulting configuration in XML format in the CLI, the ñ character maps to its equivalent UTF-8 decimal character reference `&#195;&#177;`. The same result occurs if you display the configuration in any format in a NETCONF or Junos XML protocol session.

```
[edit]
user@host# show system login user mariap | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.2R1/junos">
    <configuration junos:changed-seconds="1494033077" junos:changed-localtime="2017-05-05
18:11:17 PDT">
            <system>
                <login>
                    <user>
                        <name>mariap</name>
                        <full-name>Maria Pe&#195;&#177;a</full-name>
                        <uid>2007</uid>
                        <class>super-user</class>
                    </user>
                </login>
            </system>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

When you load configuration data onto a device, you can load non-ASCII characters using their equivalent UTF-8 decimal character references.

Configure RFC-Compliant NETCONF Sessions

IN THIS SECTION

Understanding RFC-Compliant NETCONF Sessions

When you use NETCONF to manage Junos devices, you can require that the NETCONF session enforce certain behaviors that are compliant with specific RFCs. You can configure Junos devices to be compliant with the following RFCs:

- RFC 4741, *NETCONF Configuration Protocol*
- RFC 6242, *Using the NETCONF Protocol over Secure Shell (SSH)*

To enforce RFC 4741 compliance, configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level. Configuring the `rfc-compliant` statement affects the following aspects of the NETCONF session:

- Namespaces emitted in NETCONF server replies
- NETCONF server replies for `<get>` and `<get-config>` operations in cases where the server does not return any configuration data
- NETCONF server replies that would return both an `<ok/>` element and an `<rpc-error>` element with a severity level of warning
- NETCONF server replies for `<commit>` and `<validate>` operations

To also enforce RFC 6242 compliance, configure both the `rfc-compliant` and `version-1.1` statements at the `[edit system services netconf]` hierarchy level. If you configure RFC 6242 compliance, and both peers advertise the `:base:1.1` capability in the capabilities exchange, the NETCONF session uses chunked framing instead of the end-of-message character sequence (]]>]]>) for message framing. For additional details, see "(RFC 6242) Chunked Framing" on page 149.

The session differences are described in detail in the corresponding sections.

Benefits of RFC-Compliant Sessions

- Standards Compliance: Ensures that NETCONF communications are compliant with the latest industry standards, facilitating interoperability with other compliant systems.

- Enhanced Reliability: By using chunked framing, NETCONF messages are encoded in a way that prevents misinterpretation of character sequences within XML elements, ensuring reliable message parsing.

(RFC 4741) Namespaces

By default, the NETCONF server sets the default namespace to the NETCONF namespace in the opening tag of the server's reply, and NETCONF tag names are not qualified. For example:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    ...
  </capabilities>
  <session-id>27700</session-id>
<hello>



<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/25.2R1.9/junos">
```

When you configure the `rfc-compliant` statement, the NETCONF server does not define a default namespace in its replies. Instead, the server includes a namespace declaration for the NETCONF namespace, which is bound to the `nc` prefix. The server qualifies all NETCONF tags in its replies with the prefix. If you set the default namespace to the NETCONF namespace in an RPC request, the server discards the default namespace. The server emits its reply using only the declared namespace that is bound to the `nc` prefix.

The following sample output shows the NETCONF server's `<hello>` message and capabilities exchange in an RFC-compliant NETCONF session. The `<hello>` tag contains the `xmlns:nc` declaration, and all NETCONF tags include the `nc` prefix.

```
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <nc:capabilities>
    <nc:capability>urn:ietf:params:netconf:base:1.0</nc:capability>
    ...
  </nc:capabilities>
  <nc:session-id>27703</nc:session-id>
</nc:hello>
```

The following output shows a sample RPC reply for an RFC-compliant session:

```
<nc:rpc-reply
    xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
    xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <database-status-information>
      <database-status>
        <pid>47868</pid>
        <user>admin</user>
        <terminal>pts/1</terminal>
        <start-time junos:seconds="1760133182">2025-10-10 14:53:02 PDT</start-time>
        <edit-path>{master}[edit system]</edit-path>
      </database-status>
    </database-status-information>
</nc:rpc-reply>
```

Additionally, if you request configuration data in an RFC-compliant NETCONF session, the server sets the default namespace for the `<configuration>` element to the same namespace as in the corresponding YANG model.

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>

<nc:rpc-reply
  xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
<nc:data>
<configuration
  xmlns="http://yang.juniper.net/junos/conf/root"
  junos:commit-seconds="1760133210"
  junos:commit-localtime="2025-10-10 14:53:30 PDT"
  junos:commit-user="user">
  ...
</configuration>
</nc:data>
```

```
</nc:rpc-reply>
]]>]]>
```

(RFC 4741) Changes to <get> and <get-config> Operations

The `rfc-compliant` statement affects the `<get>` and `<get-config>` server replies in cases where the server does not return any configuration data. This situation can occur, for example, when you apply a filter to return a subset of the configuration, and that portion of the configuration is empty.

In these cases, if you do not configure the `rfc-compliant` statement, the RPC reply contains an empty `<configuration>` element inside the `<data>` element.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/15.1D0/junos">
<data>
<configuration>
</configuration>
</data>
</rpc-reply>
```

If you configure the `rfc-compliant` statement, the RPC reply instead returns an empty `<data>` element and omits the `<configuration>` element.

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/25.2R1.9/junos">
<nc:data>
</nc:data>
</nc:rpc-reply>
```

(RFC 4741) <rpc-error> Elements with a Severity Level of Warning in RPC Replies

When you configure the `rfc-compliant` statement, the NETCONF server cannot return an RPC reply that includes both an `<rpc-error>` element and an `<ok/>` element. If the operation is successful, but the server reply would include one or more `<rpc-error>` elements with a severity level of warning in addition to the `<ok/>` element, then the warnings are omitted. In addition, starting in Junos OS Release 21.2R1, any warnings that are omitted during a `<commit>` operation are redirected to the system log file for tracking.

If you do not configure the `rfc-compliant` statement, the NETCONF server might issue an RPC reply that includes both an `<rpc-error>` element with a severity level of warning and an `<ok/>` element. For example, a commit operation might be successful but return a warning as in the following NETCONF server reply:

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/25.2R1.9/junos">
    <nc:rpc-error>
        <nc:error-severity>warning</nc:error-severity>
        <nc:error-message>
            uid changed for jadmin (2001->2014)
        </nc:error-message>
    </nc:rpc-error>
    <nc:ok/>
</nc:rpc-reply>
]]>]]>
```

If you configure the `rfc-compliant` statement, then the server reply omits the warning.

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/25.2R1.9/junos">
<nc:ok/>
</nc:rpc-reply>
]]>]]>
```

(RFC 4741) NETCONF Server Response to `<commit>` and `<validate>` Operations

Starting in Junos OS Release 21.2R1, when you configure the `rfc-compliant` statement, the NETCONF server's response to `<commit>` operations includes the following changes:

- If a successful `<commit>` operation returns a response with one or more warnings, the server redirects the warnings to the system log file, in addition to omitting the warnings from the response.
- The NETCONF server response emits the `<source-daemon>` element as a child of the `<error-info>` element instead of the `<rpc-error>` element.
- If you also configure the `flatten-commit-results` statement at the `[edit system services netconf]` hierarchy level, the NETCONF server emits only an `<ok/>` or `<rpc-error>` element in its response and suppresses any `<commit-results>` XML subtree.

Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, when you configure the `rfc-compliant` statement, the NETCONF server emits only an `<ok/>` or `<rpc-error>` element in response to `<validate>` operations. In earlier releases, the RPC reply also includes the `<commit-results>` element.

(RFC 6242) Chunked Framing

NETCONF sessions use a framing mechanism to separate the messages that the NETCONF server and client send within a session. By default, the NETCONF server emits the `:base:1.0` capability, and both the NETCONF server and client use the character sequence]]>]]> defined in RFC 4742 as the message separator. However, this character sequence can also potentially appear in XML attributes, comments, and processing instructions, where it could be misinterpreted as a message boundary.

To clearly define message boundaries, you can configure RFC 6242 compliance with support for chunked framing on Junos devices that support this feature. Chunked framing is a standardized framing mechanism that ensures that character sequences within XML elements are not misinterpreted as message boundaries. When you enable RFC 6242 compliance, and both peers advertise the `:base:1.1` capability in the capabilities exchange, the NETCONF session uses chunked framing for the remainder of the session.

The chunked framing mechanism encodes all NETCONF messages as chunked following the Augmented Backus-Naur Form (ABNF) rule Chunked-Message, which is defined as follows:

```
Chunked-Message = 1*chunk
                  end-of-chunks

chunk          = LF HASH chunk-size LF
                  chunk-data
chunk-size     = 1*DIGIT1 0*DIGIT
chunk-data     = 1*OCTET

end-of-chunks  = LF HASH HASH LF

DIGIT1         =  %x31-39
DIGIT          =  %x30-39
HASH           =  %x23
LF             =  %x0A
OCTET          =  %x00-FF
```

NOTE: The ABNF specification is defined in RFC 5234, *Augmented BNF for Syntax Specifications: ABNF*.

The chunk framing protocol divides NETCONF messages into distinct chunks, each with a specified size (chunk-size) and data (chunk-data), followed by an end-of-chunks marker. This structure ensures that sequences such as `]]>]]>` within XML elements are not misinterpreted as end-of-message markers.

For example, when the NETCONF client emits the `<get-system-information>` RPC using the end-of-document character sequence, it emits the message as follows:

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-system-information/>
</rpc>
]]>]]>
```

When the NETCONF client emits the same RPC using chunked framing, it might structure the message as follows:

```
#4
<rpc
#18
  message-id="102"
#85
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-system-information/>
</rpc>
##
```

Similarly, the NETCONF server returns its reply using chunked framing.

```
#141
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
xml.juniper.net/junos/24.4R1/junos"  message-id="102" >

#21
<system-information>

#90
<host-name>R1</host-name>
<hardware-model>mx960</hardware-model>
<os-name>junos</os-name>

#73
```

```
<os-version>24.4R1.10</os-version>
<serial-number>ABC123</serial-number>

#22
</system-information>

#16
</nc:rpc-reply>

##
```

To enable RFC 6242 compliance with chunked framing support:

1. Enable the NETCONF service.

   ```
   [edit system services netconf]
   user@host# set ssh
   ```

2. Configure NETCONF session compliance with RFC 4741 and RFC 6242.

   ```
   [edit system services netconf]
   user@host# set rfc-compliant
   user@host# set version-1.1
   ```

3. Commit the configuration.

   ```
   [edit system services netconf]
   user@host# commit
   ```

To use chunked framing in the NETCONF session, the client application and NETCONF server must both advertise the `:base:1.1` capability in the capabilities exchange. For example:

```
admin@host:~$ ssh 198.51.100.1 -p 830 -s netconf
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <nc:capabilities>
    <nc:capability>urn:ietf:params:netconf:base:1.1</nc:capability>
  </nc:capabilities>
</nc:hello>
]]>]]>
```

```
<!-- No zombies were killed during the creation of this user interface -->
<!-- user admin, class j-super-user -->
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
   <nc:capabilities>
    <nc:capability>urn:ietf:params:netconf:base:1.0</nc:capability>
    <nc:capability>urn:ietf:params:netconf:base:1.1</nc:capability>
    ...
  </nc:capabilities>
  <nc:session-id>80206</nc:session-id>
</nc:hello>
]]>]]>
```

After both peers advertise the `:base:1.1` capability, the NETCONF session uses chunked framing for the remainder of the session.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.2R1 and 23.4R1-EVO	Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, when you configure the `rfc-compliant` statement, the NETCONF server emits only an `<ok/>` or `<rpc-error>` element in response to `<validate>` operations. In earlier releases, the RPC reply also includes the `<commit-results>` element.
21.2R1	Starting in Junos OS Release 21.2R1, when you configure the `rfc-compliant` statement, the NETCONF server's response to `<commit>` operations is modified.
18.4R1	Starting in Junos OS Release 17.4R3, 18.2R2, 18.3R2, and 18.4R1, when you configure the `rfc-compliant` statement, the NETCONF server cannot return an RPC reply that includes both an `<rpc-error>` element and an `<ok/>` element.

RELATED DOCUMENTATION

rfc-compliant

NETCONF Monitoring

SUMMARY

You can query Junos devices to retrieve NETCONF state information and supported schemas from the NETCONF server.

IN THIS SECTION

Junos devices support concurrent management sessions from multiple local and remote NETCONF clients. At times, you need visibility into the active NETCONF sessions on a device as well as information about NETCONF server statistics and support. Having easy access to NETCONF state information enables you to more effectively manage your network devices.

The NETCONF monitoring data model provides operational information about the NETCONF server. NETCONF clients can query a Junos device to retrieve NETCONF state information from the NETCONF server. Clients can request information for NETCONF capabilities, NETCONF sessions and statistics, configuration datastores, and supported schemas.

For more information about the NETCONF monitoring model, see RFC 6022, *YANG Module for NETCONF Monitoring*.

NETCONF State Information Overview

The NETCONF monitoring data model defines the NETCONF server's operational data. The `netconf-state` container comprises subtrees that define and include the data for the different areas of operation.

Table 9 on page 153 outlines the `netconf-state` subtrees supported on Junos devices.

Table 9: Supported netconf-state Subtrees

`netconf-state` Subtree	Description
`capabilities`	NETCONF operations supported by the NETCONF server.

Table 9: Supported netconf-state Subtrees *(Continued)*

`netconf-state` Subtree	Description
`datastores`	Available configuration datastores, for example, `candidate` or `running` (active), and their lock state.
`schemas`	Schemas supported on the device.
`sessions`	Active NETCONF management sessions on the device.
`statistics`	NETCONF server performance data.

Junos devices that support the NETCONF monitoring data model advertise this capability in the NETCONF session's capabilities exchange during session setup.

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    ...
    <capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
    ...
  </capabilities>
  <session-id>12976</session-id>
</hello>
]]>]]>
```

To request NETCONF state information, send a `<get>` request, and specify the `netconf-state` subtree of interest, for example, `<datastores>`.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <subtree>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

When you request NETCONF state information, the server's RPC reply includes the `<data>` and `<netconf-state>` elements. These elements enclose the subtree for the requested information.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
...
</netconf-state>
</data>
</rpc-reply>
```

The `netconf-state` `<sessions>` and `<statistics>` subtrees include information about active NETCONF sessions and NETCONF server data, respectively. Table 10 on page 155 outlines the supported elements returned for these filters. The `<sessions>` data includes per-session counters. The `<statistics>` data reports global counters for the NETCONF server.

Table 10: NETCONF Sessions and Statistics Data

Node	Description	Filters
`<dropped-sessions>`	Number of NETCONF sessions that were abnormally terminated.	`<statistics>`

`<in-bad-rpcs>`	Number of incorrect RPC messages received by the server.	`<sessions>`
		`<statistics>`

`<in-rpcs>`	Number of correct RPC messages received by the server.	`<sessions>`
		`<statistics>`

`<in-sessions>`	Number of NETCONF sessions started.	`<statistics>`
`<login-time>`	Date and time when the NETCONF session was established.	`<sessions>`
`<netconf-start-time>`	Date and time when the NETCONF server was started.	`<statistics>`

Table 10: NETCONF Sessions and Statistics Data *(Continued)*

Node	Description	Filters
`<out-notifications>`	Number of `<notification>` messages sent.	`<sessions>` `<statistics>`
`<out-rpc-errors>`	Number of NETCONF server RPC replies that contained an `<rpc-error>` element.	`<sessions>` `<statistics>`
`<session-id>`	NETCONF session identifier.	`<sessions>`
`<source-host>`	IP address or hostname from which the NETCONF client connected.	`<sessions>`
`<transport>`	Transport protocol for the NETCONF session, for example, `netconf-ssh`.	`<sessions>`
`<username>`	Client identity authenticated by the NETCONF transport protocol.	`<sessions>`

Retrieve NETCONF Capabilities

A NETCONF client can retrieve the NETCONF server's capabilities. The capabilities define the operations supported by the NETCONF server. The NETCONF server advertises the supported capabilities during session setup. By default, Junos devices do not advertise supported YANG modules in the capabilities list. However, you can configure the device to include them.

To request the capabilities of the NETCONF server:

1. (Optional) Configure the device to advertise the different supported YANG modules in the NETCONF capabilities list, if desired.

```
[edit system services netconf hello-message yang-module-capabilities]
user@host# set advertise-custom-yang-modules
user@host# set advertise-native-yang-modules
user@host# set advertise-standard-yang-modules
user@host# commit and-quit
```

2. In a NETCONF session, execute a `<get>` operation for the `netconf-state/capabilities` subtree.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <capabilities/>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

The NETCONF server returns the `<capabilities>` element with the supported capabilities. The `<capabilities>` information is identical to that in the `<hello>` message exchange sent during session setup.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file</capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?scheme=http,ftp,file</
capability>
    <capability>urn:ietf:params:xml:ns:yang:ietf-yang-metadata?module=ietf-yang-
metadata&amp;revision=2016-08-05</capability>
    <capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
    <capability>http://yang.juniper.net/junos/jcmd?module=junos-configuration-
metadata&amp;revision=2021-09-01</capability>
    <capability>http://yang.juniper.net/junos/common/types?module=junos-common-
types&amp;revision=2023-01-01</capability>
    <capability>http://yang.juniper.net/junos/conf/access-profile?module=junos-conf-access-
```

```
profile&amp;revision=2023-01-01</capability>
    <capability>http://yang.juniper.net/junos/conf/access?module=junos-conf-
access&amp;revision=2023-01-01</capability>
    <capability>http://yang.juniper.net/junos/conf/accounting-options?module=junos-conf-
accounting-options&amp;revision=2023-01-01</capability>
    ...
  </capabilities>
</netconf-state>
</data>
</rpc-reply>
```

Retrieve Configuration Datastores

The configuration datastores are the configuration databases supported on the device. When you request information about the configuration datastores, the server also returns their lock status.

To request the list of configuration datastores supported by the NETCONF server:

- In a NETCONF session, execute a `<get>` operation for the `netconf-state/datastores` subtree.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <datastores/>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

The NETCONF server returns the configuration datastores and their lock states. In this case, the datastores include the `candidate` configuration, which has a lock on it, and the `running` (active) configuration.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
  <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
    <datastores>
      <datastore>
        <name>candidate</name>
        <locks>
```

```
            <locked-by-session>0</locked-by-session>
            <locked-time junos:seconds="1691539727">2023-08-08T17:08:47-07:00</locked-time>
          </locks>
        </datastore>
        <datastore>
          <name>running</name>
        </datastore>
      </datastores>
    </netconf-state>
  </data>
  </rpc-reply>
```

Retrieve Schemas

NETCONF clients can request the list of schemas supported on the device. By default, Junos devices return only the Junos native schemas in the supported schemas list. However, you can configure the device to include any additional supported schemas, including custom YANG modules that are installed on the device as well as standard modules, such as OpenConfig.

To request the list of supported schemas:

1. (Optional) Configure the device to emit any additional schemas, other than the default native schemas.

   - To include schemas for custom YANG modules installed on the device, configure the `retrieve-custom-yang-modules` statement.

     ```
     [edit system services netconf netconf-monitoring]
     user@host# set netconf-state-schemas retrieve-custom-yang-modules
     ```

   - To include schemas for standard YANG modules, such as OpenConfig, configure the `retrieve-standard-yang-modules` statement.

     ```
     [edit system services netconf netconf-monitoring]
     user@host# set netconf-state-schemas retrieve-standard-yang-modules
     ```

2. If you modified the configuration in the previous step, commit the configuration.

   ```
   [edit]
   user@host# commit and-quit
   ```

3. In a NETCONF session, execute a `<get>` operation for the `netconf-state/schemas` subtree.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <schemas/>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

The device returns the list of supported schemas. The output includes the Junos native schemas. The output also include custom and standard schemas, if you configured the device to emit these schemas.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
<schemas>
<schema>
<identifier>junos-common-types</identifier>
<version>2023-01-01</version>
<format>yang</format>
<namespace>http://yang.juniper.net/junos/common/types</namespace>
<location>NETCONF</location>
</schema>
<schema>
<identifier>junos-conf-access-profile</identifier>
<version>2023-01-01</version>
<format>yang</format>
<namespace>http://yang.juniper.net/junos/conf/access-profile</namespace>
<location>NETCONF</location>
</schema>
...
</schemas>
</netconf-state>
</data>
</rpc-reply>
```

The `netconf-state/schemas` subtree only returns the identifiers for the supported schemas. It does not include the actual schemas. Given the identifer, you can retrieve a specific schema instance. To request a schema instance in a NETCONF session:

- Execute the `<get-schema>` operation and specify the schema identifier.

  ```
  <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-schema xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
      <identifier>schema-identifier</identifier>
    </get-schema>
  </rpc>
  ```

  For example, the following RPC retrieves the `junos-conf-access-profile` schema.

  ```
  <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-schema xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
      <identifier>junos-conf-access-profile</identifier>
    </get-schema>
  </rpc>
  ```

The NETCONF server returns the schema in YANG format, which is the default and only supported format.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
/*
 * Copyright (c) 2023 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-conf-access-profile {
   namespace "http://yang.juniper.net/junos/conf/access-profile";

   prefix jc-access-profile;

   import junos-common-types {
     prefix jt;
     revision-date 2023-01-01;
   }

   import junos-conf-root {
```

```
      prefix jc;
      revision-date 2023-01-01;
    }

    organization "Juniper Networks, Inc.";

    contact "yang-support@juniper.net";

    description "Junos access-profile configuration module";

    revision 2023-01-01 {
      description "Junos: 23.4R1.12-EVO";
    }

    augment /jc:configuration {
      uses access-profile-group;
    }
    augment /jc:configuration/jc:groups {
      uses access-profile-group;
    }
    grouping access-profile-group {
      container access-profile {
        description "Access profile for this instance";
        leaf access-profile-name {
          description "Profile name";
          type string;
        }
      }
    }
  }
</data>
</rpc-reply>
```

Retrieve NETCONF Session Information

NETCONF clients can request a list of the active NETCONF sessions on the device. The NETCONF server returns the active sessions along with information about each session. The returned data includes per-session counters. See Table 10 on page 155 for descriptions of the output fields.

For sessions where certain values are undefined, for example, internal sessions, the default values for `transport`, `username`, and `source-host` are `netconf-ssh`, `internal-user`, and `local-host`, respectively.

To retrieve the active NETCONF sessions on the device:

- In a NETCONF session, execute a `<get>` operation for the `netconf-state/sessions` subtree.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <sessions/>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

The NETCONF server returns the active NETCONF sessions along with the session-specific data.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
  <sessions>
    <session>
      <session-id>2614</session-id>
      <transport>netconf-ssh</transport>
      <username>admin</username>
      <source-host>10.1.1.101</source-host>
      <login-time junos:seconds="1691699108">2023-08-10T13:25:08-07:00</login-time>
      <in-rpcs>6</in-rpcs>
      <in-bad-rpcs>0</in-bad-rpcs>
      <out-rpc-errors>1</out-rpc-errors>
      <out-notifications>0</out-notifications>
    </session>
    <session>
      <session-id>2879</session-id>
      <transport>netconf-ssh</transport>
      <username>sec-admin</username>
      <source-host>198.51.100.11</source-host>
      <login-time junos:seconds="1691699237">2023-08-10T13:27:17-07:00</login-time>
      <in-rpcs>11</in-rpcs>
      <in-bad-rpcs>1</in-bad-rpcs>
      <out-rpc-errors>2</out-rpc-errors>
      <out-notifications>0</out-notifications>
    </session>
    <session>
      <session-id>13559</session-id>
      <transport>netconf-ssh</transport>
      <username>root</username>
      <source-host>local-host</source-host>
      <login-time junos:seconds="1689712208">2023-07-18T13:30:08-07:00</login-time>
      <in-rpcs>14</in-rpcs>
      <in-bad-rpcs>0</in-bad-rpcs>
      <out-rpc-errors>0</out-rpc-errors>
      <out-notifications>0</out-notifications>
    </session>
  </sessions>
</netconf-state>
</data>
</rpc-reply>
```

Retrieve NETCONF Server Statistics

A NETCONF client can request the NETCONF server statistics for a given device. Whereas the `<netconf-state> <sessions>` filter returns per-session counters, the `<netconf-state> <statistics>` filter returns global counters for the NETCONF server. See Table 10 on page 155 for descriptions of the output fields.

To request NETCONF server statistics on a device:

- In a NETCONF session, execute a `<get>` operation for the `netconf-state/statistics` subtree.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <statistics/>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

The NETCONF server returns the global performance data for the server.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.12-EVO/junos"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
<statistics>
```

```
<netconf-start-time junos:seconds="1689712174">2023-07-18T13:29:34-07:00</netconf-start-time>
<in-sessions>43</in-sessions>
<dropped-sessions>3</dropped-sessions>
<in-rpcs>58</in-rpcs>
<in-bad-rpcs>48</in-bad-rpcs>
<out-rpc-errors>4</out-rpc-errors>
<out-notifications>2</out-notifications>
</statistics>
</netconf-state>
</data>
</rpc-reply>
```

RELATED DOCUMENTATION

<get> | **195**

NETCONF Event Notifications

SUMMARY

NETCONF clients can subscribe to event notifications in NETCONF sessions to receive alerts for events that might impact device operations or management activities.

IN THIS SECTION

NETCONF Event Notifications Overview

Certain devices running Junos OS Evolved support NETCONF event notifications, an asynchronous event notification service between a NETCONF server and a NETCONF client. After you enable the notification service, the NETCONF server sends event notifications, asynchronously as the events occur, to all NETCONF clients that subscribe to the notifications. Clients can subscribe to NETCONF notifications to receive alerts for events that might impact device operations or management activities.

The NETCONF server sends notifications for the following types of events:

- `netconf-session-start`—Indicates when a NETCONF session starts and identifies the user who started the session.
- `netconf-session-end`—Indicates when a NETCONF session ends and identifies the user who owned the session and the reason that the session was terminated.
- `netconf-config-change`—Indicates when a management session commits changes to the active configuration and provides a summary of the changes.

You can enable the NETCONF event notification service on supported devices. See "How to Enable and Subscribe to NETCONF Event Notifications" on page 170 for instructions. You can optionally configure the interleave capability, which enables NETCONF clients to subscribe to notifications and send RPCs in the same NETCONF session, as described in "Interleave Capability" on page 168.

NOTE: We recommend that you configure and stream only one set of notifications, either NETCONF notifications or telemetry data, at a time. If you configure a device to enable NETCONF notifications, the device streams all notifications over the NETCONF channel and does not stream telemetry notifications over gRPC.

After you enable NETCONF event notifications, the NETCONF server advertises the `notification` capability and the `interleave` capability in the capabilities exchange.

```
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
   <nc:capabilities>
    ...
    <nc:capability>urn:ietf:params:netconf:capability:notification:1.0</nc:capability>
    <nc:capability>urn:ietf:params:xml:ns:netconf:notification:1.0</nc:capability>
    <nc:capability>urn:ietf:params:netconf:capability:interleave:1.0</nc:capability>
    <nc:capability>urn:ietf:params:xml:ns:netmod:notification</nc:capability>
    ...
  </nc:capabilities>
  <nc:session-id>29862</nc:session-id>
</nc:hello>
```

To subscribe to the notification service for events on a specific device, a NETCONF client sends a `<create-subscription>` RPC to the NETCONF server on the device and indicates the following:

- `<stream>`—The stream of events that is of interest.

  A stream is a set of event notifications that matches some forwarding criteria. A subscription is bound to a single stream for the lifetime of the subscription. The `NETCONF` stream is the default and only supported stream on Junos devices. The NETCONF server returns an error if the subscription request is for any other stream. If you omit this parameter, the device treats the subscription request as a request for the `NETCONF` stream.

- `<filter>`—A subtree filter that selects a subset of all possible events.

  If a NETCONF client specifies a filter, the server forwards only those events selected by the filter. If a client does not specify a filter, the server forwards all events. Junos OS Evolved supports only top-level filters and does not support using XPATH filters for this element. See "Filtering Capability" on page 169 for more information.

After a NETCONF client subscribes to event notifications, the NETCONF server sends the notifications as they occur. The notifications continue until the NETCONF session terminates.

NOTE: A NETCONF client receives all event notifications by default. There is no way to restrict or limit the content of a notification based on user privileges. Because some events, for example, `netconf-config-change` events, can contain sensitive information, it is important to control read access to the information.

For additional information about NETCONF event notifications, see the following RFCs:

- RFC 5277, *NETCONF Event Notifications*
- RFC 6470, *Network Configuration Protocol (NETCONF) Base Notifications*

NETCONF Event Notification Format

NETCONF event notifications are well-formed XML documents. When the NETCONF server receives an internal event, it converts it to an appropriate XML encoding with a top-level `<notification>` element and an `<eventTime>` child element. The actual content contained in the notification depends on the event.

A subscription request can include filters for specific types of notifications. If the subscription request includes filters, the user-defined filters are applied to each notification in the event stream, and the NETCONF server forwards only matching events to the client.

The following sample event notification contains a `netconf-config-change` event. The notification captures the event timestamp, the commit timestamp, the user who committed the configuration changes, and a summary of those changes.

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
    <eventTime>2021-04-15T11:39:41-07:00</eventTime>
    <netconf-config-change xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-notifications">
        <change-time>2021-04-15T18:39:41Z</change-time>
        <changed-by>
```

```
            <username>admin</username>
            <session-id>29862</session-id>
            <source-host>198.51.100.25</source-host>
        </changed-by>
        <datastore>running</datastore>
        <edit>
            <target xmlns:junos-conf-root="http://yang.juniper.net/junos/conf/root" xmlns:junos-
conf-interfaces="http://yang.juniper.net/junos/conf/interfaces">/junos-conf-root:configuration/
junos-conf-interfaces:interfaces/junos-conf-interfaces:interface[junos-conf-
interfaces:name='et-0/0/0']/junos-conf-interfaces:description</target>
            <operation>replace</operation>
        </edit>
    </netconf-config-change>
</notification>
```

The following notifications contain sample `netconf-session-start` and `netconf-session-end` events:

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
    <eventTime>2021-04-15T11:28:51-07:00</eventTime>
    <netconf-session-start xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-notifications">
        <username>admin</username>
        <session-id>29862</session-id>
        <source-host>198.51.100.25</source-host>
    </netconf-session-start>
</notification>
```

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
    <eventTime>2021-04-15T11:49:06-07:00</eventTime>
    <netconf-session-end xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-notifications">
        <username>admin</username>
        <session-id>29862</session-id>
        <source-host>198.51.100.25</source-host>
        <termination-reason>closed</termination-reason>
    </netconf-session-end>
</notification>
```

Interleave Capability

By default, after a NETCONF client subscribes to event notifications in a NETCONF session, the client cannot also send RPCs in the same session. The interleave capability enables the NETCONF client and server to continue exchanging RPCs and RPC replies within the same NETCONF session that is used for notifications. The interleave capability reduces the overall number of NETCONF sessions because you don't need a dedicated NETCONF session for notifications. To use the interleave capability on devices that support this feature, you must configure the `interleave` statement at the `[edit system services netconf notification]` hierarchy level.

A NETCONF client ends a subscription by terminating either the NETCONF session or the NETCONF session's underlying transport session, for example, with a `<close-session>` or `<kill-session>` operation. If you enable the interleave capability, a client can terminate the session by executing a `<close-session>` operation from within the same session. If you do not enable the interleave capability, the client can terminate the session, for example, by executing the `<kill-session>` operation from another session.

Filtering Capability

When a NETCONF client subscribes to event notifications, the client can subscribe to all event notifications in the stream, or the client can subscribe to a subset of event notifications. To subscribe to a subset of event notifications, the client includes the optional `<filter>` element in the `<create-subscription>` RPC. If the subscription request includes filters, the filters are applied to each notification in the event stream, and the NETCONF server forwards only matching events to the client. Otherwise, the server forwards all events.

The `<filter>` parameter format for the `<create-subscription>` operation is similar to the `filter` parameter format for other NETCONF operations. It encloses a subtree filter that selects the desired event notifications. In the case of the `<create-subscription>` operation, however, Junos devices only support subtree filters that match against the top-level element of the enclosed notification, for example, `<netconf-config-change>`. The filter is applied to the notifications in the stream and only against the contents of the `<notification>` wrapper.

NOTE: Junos devices do not support using XPath to filter the notifications.

When you filter for specific notifications, you must include the appropriate namespace in the tag. If you do not specify the namespace, the event notifications will not match the filter, and the NETCONF server will not forward the notifications. In the following example, the subscription request returns a subset of all NETCONF event notifications. The filter selects and forwards only `<netconf-config-change>` events and `<oc-ifl-event>` events.

```
<nc:rpc xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <create-subscription xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
    <filter type="subtree">
      <netconf-config-change xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-notifications"></
netconf-config-change>
      <oc-ifl-event xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0"></oc-ifl-event>
    </filter>
  </create-subscription>
</nc:rpc>
]]>]]>
```

How to Enable and Subscribe to NETCONF Event Notifications

IN THIS SECTION

You must enable the NETCONF event notification service on a device before a NETCONF client can subscribe to event notifications in a NETCONF session. After the service is enabled, a NETCONF client subscribes to receive event notifications by sending a subscription request to the NETCONF server. The NETCONF server reply indicates if the request is successful. If the request is successful, the server sends asynchronous event notifications to the NETCONF client as the events occur and until the NETCONF session is terminated.

This example requires the following hardware and software:

- Device running Junos OS Evolved Release 21.2R1 or later that supports the NETCONF event notification service. See Feature Explorer for supported devices.

To enable and subscribe to NETCONF event notifications, perform the following tasks:

Enable the NETCONF Event Notification Service

To enable a client to subscribe to event notifications in a NETCONF session:

1. Enable the NETCONF event notification service by configuring the `notification` statement.

```
[edit]
user@host# set system services netconf notification
```

2. (Optional) Configure the `interleave` option to enable a NETCONF client to execute RPCs in a NETCONF session that also subscribes to notifications.

   ```
   [edit]
   user@host# set system services netconf notification interleave
   ```

3. Configure the `rfc-compliant` statement to ensure the device is compliant with NETCONF RFC 4741.

   ```
   [edit]
   user@host# set system services netconf rfc-compliant
   ```

4. (Optional) Configure NETCONF tracing options for troubleshooting any issues.

   ```
   [edit]
   user@host# set system services netconf traceoptions file filename
   user@host# set system services netconf traceoptions file files number size size
   user@host# set system services netconf traceoptions flag flag
   ```

   For example:

   ```
   [edit]
   user@host# set system services netconf traceoptions file netconf.log
   user@host# set system services netconf traceoptions file files 3 size 3m
   user@host# set system services netconf traceoptions flag all
   ```

5. Enable notification services on the default port for applications running on the device.

   In releases that require the `allow-clients` statement, you must also specify the clients (hostnames or IP addresses) that are allowed to subscribe to notifications.

   ```
   [edit]
   user@host# set system services extension-service notification allow-clients address [address1
   address2]
   ```

   For example:

   ```
   [edit]
   user@host# set system services extension-service notification allow-clients address
   [198.51.100.25 10.1.1.101]
   ```

6. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

Subscribe to Receive Event Notifications

After you enable the NETCONF event notification service on a device, NETCONF clients can subscribe to receive event notifications in a NETCONF session. A NETCONF client can include the following optional parameters in the subscription request:

- `<stream>`—Stream of events that is of interest. The default and only acceptable value is `NETCONF`.
- `<filter>`—Subtree filter that selects a subset of all possible events.

To subscribe to event notifications in a NETCONF session:

1. Start the NETCONF session.
2. Verify that the NETCONF event notification service is enabled on the device by confirming that the notification capability is advertised in the capabilities exchange.

```
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
   <nc:capabilities>
    ...
    <nc:capability>urn:ietf:params:netconf:capability:notification:1.0</nc:capability>
    <nc:capability>urn:ietf:params:xml:ns:netconf:notification:1.0</nc:capability>
    <nc:capability>urn:ietf:params:xml:ns:netmod:notification</nc:capability>
    <nc:capability>urn:ietf:params:netconf:capability:interleave:1.0</capability>
  </nc:capabilities>
  <nc:session-id>29862</nc:session-id>
</nc:hello>
]]>]]>
```

3. Issue a `create-subscription` request, and optionally specify the `NETCONF` stream.

   - To subscribe to all notifications, omit the `<filter>` parameter.

```
<nc:rpc xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
<create-subscription xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
<stream>NETCONF</stream>
```

```
</create-subscription>
</nc:rpc>
```

- To subscribe to a subset of notifications, include the `<filter type="subtree">` element, and define one or more subtree filters for the notifications of interest. Junos devices only support subtree filters that match against the top-level element of the enclosed notification. You must include the appropriate namespace in the tag for the specific notification.

  ```
  <nc:rpc xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <create-subscription xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <stream>NETCONF</stream>
      <filter type="subtree">
        ...subtree filters...
      </filter>
    </create-subscription>
  </nc:rpc>
  ```

  For example:

  ```
  <nc:rpc xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <create-subscription xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <filter type="subtree">
        <netconf-config-change xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-
  notifications"></netconf-config-change>
      </filter>
    </create-subscription>
  </nc:rpc>
  ]]>]]>
  ```

4. Verify that the subscription request is successful.

   The NETCONF server returns `<ok/>` if the request is successful or an `<rpc-error>` element if it cannot complete the subscription request.

   ```
   <nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
   xml.juniper.net/junos/21.2R1/junos">
   <nc:ok/>
   </nc:rpc-reply>
   ]]>]]>
   ```

If the subscription request is successful, the NETCONF server starts sending event notifications asynchronously over the connection.

5. If the interleave capability is enabled, the NETCONF client can continue to send RPCs within the same session.

Terminate the Subscription

A NETCONF client terminates a subscription to receive event notifications by terminating either the NETCONF session or the NETCONF session's underlying transport session.

To terminate the NETCONF session and subscription, perform one of the following actions:

- If the interleave capability is enabled, issue the `<close-session/>` operation in the NETCONF session.

  ```
  <rpc><close-session/></rpc>
  ```

- Issue the `<kill-session>` operation from an external NETCONF session, and specify the session ID for the NETCONF session to end (as defined in the `<session-id>` element of the initial `<hello>` exchange).

  ```
  <rpc><kill-session><session-id>29862</session-id></kill-session></rpc>
  ```

- Terminate the NETCONF session's underlying transport session.

CHAPTER 5

NETCONF Tracing Operations

IN THIS CHAPTER

NETCONF and Junos XML Protocol Tracing Operations Overview

You can configure tracing operations for the NETCONF and Junos XML management protocols. NETCONF and Junos XML protocol tracing operations record NETCONF and Junos XML protocol session data, respectively, in a trace file. By default, devices running Junos OS and devices running Junos OS Evolved do not enable NETCONF or Junos XML protocol tracing operations.

You configure NETCONF and Junos XML protocol tracing operations at the `[edit system services netconf traceoptions]` hierarchy level. When you enable tracing operations, the configuration applies to both NETCONF and Junos XML protocol sessions. The system adds the `[NETCONF]` or `[JUNOScript]` tag to the log file entries to distinguish the session type.

```
[edit system services]
netconf {
    traceoptions {
        file <filename> <files number> <match regular-expression> <size size> <world-readable |
 no-world-readable>;
        flag flag;
        no-remote-trace;
        on-demand;
    }
}
```

To enable tracing operations and trace all incoming and outgoing data from NETCONF and Junos XML protocol sessions on a device, configure the `flag all` statement. You can also configure the `flag debug` statement to enable debug-level tracing. However, we recommend using the `flag all` option.

You can restrict tracing to only incoming or outgoing session data by configuring the flag value as either `incoming` or `outgoing`, respectively. Additionally, you can restrict the trace output to include only those lines that match a particular expression. To use specific match criteria, configure the `match` statement and define the regular expression against which to match the output.

To control the tracing operation from within a NETCONF or Junos XML protocol session, configure the `on-demand` statement. This option requires that you start and stop trace operations from within the session. To start tracing for that session, issue the following RPC within the session:

```
<rpc><request-netconf-trace><start/></request-netconf-trace></rpc>
```

To stop tracing for that session, issue the following RPC:

```
<rpc><request-netconf-trace><stop/></request-netconf-trace></rpc>
```

NETCONF and Junos XML protocol tracing operations record session data in the file **/var/log/netconf**. To specify a different trace file, configure the `file` statement and the filename.

By default, when the trace file reaches 128 KB in size, it is compressed and renamed to ***filename*.0.gz**, then ***filename*.1.gz**, and so on, until there are 10 trace files. Then the oldest trace file (***filename*.9.gz**) is overwritten. You can configure limits on the number and size of trace files by including the `file files` *`number`* and `file size` *`size`* statements. You can configure up to a maximum of 1000 files. Specify the file size in bytes or use *`size`*`k` to specify KB, *`size`*`m` to specify MB, or *`size`*`g` to specify GB. You cannot configure the maximum number of trace files and the maximum trace file size independently. If you configure one option, you must also configure the other option along with a filename.

By default, access to the trace file is restricted to the owner. You can configure access by including either the `world-readable` or `no-world-readable` statement. The `no-world-readable` statement, which is the default, restricts trace file access to the owner. The `world-readable` statement enables unrestricted access to the trace file.

RELATED DOCUMENTATION

Example: Trace NETCONF and Junos XML Protocol Session Operations

netconf

ssh (NETCONF)

traceoptions (NETCONF and Junos XML Protocol)

Example: Trace NETCONF and Junos XML Protocol Session Operations

IN THIS SECTION

This example configures tracing operations for NETCONF and Junos XML protocol sessions.

Requirements

- A device running Junos OS or a device running Junos OS Evolved.

Overview

This example configures basic tracing operations for NETCONF and Junos XML protocol sessions. When you configure tracing operations at the `[edit system services netconf traceoptions]` hierarchy, the device enables tracing operations for both NETCONF and Junos XML protocol sessions. The system adds the `[NETCONF]` or `[JUNOScript]` tag to the log file entries to distinguish the session type.

In this example, you configure the trace file **netconf-ops.log**. You configure a maximum number of 20 trace files and a maximum size of 3 MB for each file. The `flag all` statement configures tracing for all incoming and outgoing NETCONF and Junos XML protocol data. The `world-readable` option enables unrestricted access to the trace files.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level.

```
set system services netconf ssh
set system services netconf traceoptions file netconf-ops.log
set system services netconf traceoptions file size 3m
set system services netconf traceoptions file files 20
set system services netconf traceoptions file world-readable
set system services netconf traceoptions flag all
```

Configure NETCONF and Junos XML Protocol Tracing Operations

Step-by-Step Procedure

To configure NETCONF and Junos XML protocol tracing operations:

1. For NETCONF sessions, enable NETCONF over SSH.

   ```
   [edit]
   user@R1# set system services netconf ssh
   ```

2. Configure the traceoptions flag to specify which session data to capture.

   You can specify incoming, outgoing, all, or debug data. This example configures tracing for all session data.

   ```
   [edit]
   user@R1# set system services netconf traceoptions flag all
   ```

3. (Optional) Configure the filename of the trace file.

   The following statement configures the trace file **/var/log/netconf-ops.log**. If you do not specify a filename, the system logs NETCONF and Junos XML protocol session data in **/var/log/netconf**.

   ```
   [edit]
   user@R1# set system services netconf traceoptions file netconf-ops.log
   ```

4. (Optional) Configure the maximum number of trace files and the maximum size of each file.

   The following statements configure a maximum of 20 trace files with a maximum size of 3 MB per file.

   ```
   [edit]
   user@R1# set system services netconf traceoptions file files 20
   user@R1# set system services netconf traceoptions file size 3m
   ```

5. (Optional) Restrict the trace output to include only those lines that match a particular regular expression.

   The following configuration, which is not used in this example, matches on and logs only session data that contains “error-message”.

   ```
   [edit]
   user@R1# set system services netconf traceoptions file match error-message
   ```

6. (Optional) Configure on-demand tracing to control tracing operations from the NETCONF or Junos XML protocol session.

   The following configuration, which is not used in this example, enables on-demand tracing.

   ```
   [edit]
   user@R1# set system services netconf traceoptions on-demand
   ```

7. (Optional) Configure the permissions on the trace file by specifying whether the file is `world-readable` or `no-world-readable`.

   This example enables unrestricted access to the trace file.

   ```
   [edit]
   user@R1# set system services netconf traceoptions file world-readable
   ```

8. Commit the configuration.

   ```
   [edit]
   user@R1# commit
   ```

Results

```
[edit]
system {
    services {
        netconf {
            ssh;
            traceoptions {
                file netconf-ops.log size 3m files 20 world-readable;
                flag all;
            }
        }
    }
}
```

Verification

IN THIS SECTION

Verify NETCONF and Junos XML Protocol Tracing Operation

Purpose

Verify that the device logs NETCONF and Junos XML protocol operations to the configured trace file. This example logs both incoming and outgoing NETCONF and Junos XML protocol data. In the sample NETCONF session, which is not detailed here, the user modifies the candidate configuration to include the **bgp-troubleshoot.slax** op script and then commits the configuration.

Action

Display the configured trace file by issuing the **show log** ***filename*** operational mode command.

```
user@R1 show log netconf-ops.log
Apr  3 13:09:04 [NETCONF] Started tracing session: 3694
Apr  3 13:09:29 [NETCONF] - [3694] Incoming: <rpc>
Apr  3 13:09:29 [NETCONF] - [3694] Outgoing: <rpc-reply
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/junos/24.4R1/
junos">
Apr  3 13:09:39 [NETCONF] - [3694] Incoming: <edit-config>
Apr  3 13:09:43 [NETCONF] - [3694] Incoming: <target>
Apr  3 13:09:47 [NETCONF] - [3694] Incoming: <candidate/>
Apr  3 13:09:53 [NETCONF] - [3694] Incoming: </target>
Apr  3 13:10:07 [NETCONF] - [3694] Incoming: <default-operation>merge</default-operation>
Apr  3 13:10:10 [NETCONF] - [3694] Incoming: <config>
Apr  3 13:10:13 [NETCONF] - [3694] Incoming: <configuration>
Apr  3 13:10:16 [NETCONF] - [3694] Incoming: <system>
Apr  3 13:10:19 [NETCONF] - [3694] Incoming: <scripts>
Apr  3 13:10:23 [NETCONF] - [3694] Incoming: <op>
Apr  3 13:10:26 [NETCONF] - [3694] Incoming: <file>
Apr  3 13:10:44 [NETCONF] - [3694] Incoming: <name>bgp-troubleshoot.slax</name>
Apr  3 13:10:46 [NETCONF] - [3694] Incoming: </file>
Apr  3 13:10:48 [NETCONF] - [3694] Incoming: </op>
Apr  3 13:10:52 [NETCONF] - [3694] Incoming: </scripts>
Apr  3 13:10:56 [NETCONF] - [3694] Incoming: </system>
Apr  3 13:11:00 [NETCONF] - [3694] Incoming: </configuration>
Apr  3 13:11:00 [NETCONF] - [3694] Outgoing: <ok/>
Apr  3 13:11:12 [NETCONF] - [3694] Incoming: </config>
Apr  3 13:11:18 [NETCONF] - [3694] Incoming: </edit-config>
Apr  3 13:11:26 [NETCONF] - [3694] Incoming: </rpc>
Apr  3 13:11:26 [NETCONF] - [3694] Outgoing: </rpc-reply>
Apr  3 13:11:26 [NETCONF] - [3694] Outgoing: ]]>]]>
Apr  3 13:11:31 [NETCONF] - [3694] Incoming: ]]>]]>

Apr  3 13:14:20 [NETCONF] - [3694] Incoming: <rpc>
Apr  3 13:14:20 [NETCONF] - [3694] Outgoing: <rpc-reply
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/junos/24.4R1/
junos">
Apr  3 13:14:26 [NETCONF] - [3694] Incoming: <commit/>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: <ok/>
Apr  3 13:14:35 [NETCONF] - [3694] Incoming: </rpc>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: </rpc-reply>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: ]]>]]>
Apr  3 13:14:40 [NETCONF] - [3694] Incoming: ]]>]]>

Apr  3 13:30:48 [NETCONF] - [3694] Outgoing: <!-- session end at 2025-04-03 13:30:48 PDT -->
```

Meaning

This example configures the `flag all` statement, so the trace file logs all incoming and outgoing data for any NETCONF and Junos XML protocol sessions. Each operation includes the date and timestamp. The log file indicates the session type, either NETCONF or Junos XML protocol, by including the `[NETCONF]` or `[JUNOScript]` tag, respectively. The device distinguishes multiple NETCONF and Junos XML protocol sessions by using a unique session number. In this example, only one NETCONF session, using session identifier 3694, is active.

RELATED DOCUMENTATION

NETCONF and Junos XML Protocol Tracing Operations Overview

traceoptions (NETCONF and Junos XML Protocol)

CHAPTER 6

NETCONF Protocol Operations and Attributes

IN THIS CHAPTER

<close-session/>

IN THIS SECTION

Usage

```
<rpc>
    <close-session/>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server end the current session.

RELATED DOCUMENTATION

End a NETCONF Session and Close the Connection | 136

<commit>

IN THIS SECTION

Usage

```
<rpc>
    <commit/>
</rpc>
]]>]]>
```

```
<rpc>
    <commit>
```

```
        <confirmed/>
        <confirm-timeout>rollback-delay</confirm-timeout>
    </commit>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server perform one of the variants of the commit operation on the candidate configuration or open configuration database. Table 11 on page 185 describes the commit operations.

NOTE: The `<confirmed/>` tag is not supported when committing configuration data to the ephemeral configuration database.

Table 11: Commit Operations

`<commit>` Operation	Description
`<commit/>`	Commit the configuration immediately, making it the active configuration on the device.

`<commit>`
    `<confirmed/>`
`</commit>`
 	
`<commit>`
    `<confirmed/>`
    `<confirm-timeout>`*rollback-delay*`</confirm-timeout>`
`</commit>`
	Commit the configuration but require an explicit confirmation for the commit to become permanent. If the commit is not confirmed, the configuration rolls back to the previous configuration after the specified time.
	
	Optionally, include the `<confirm-timeout>` element to specify the rollback delay in the range from 1 through 4,294,967,295 seconds. By default, the rollback occurs after 600 seconds.
	
	To delay the rollback again (past the original rollback deadline), emit the `<commit><confirmed/></commit>` tags before the deadline passes, and optionally include the `<confirm-timeout>` element. The rollback can be delayed repeatedly in this way.
	
	To confirm the commit, emit the empty `<commit/>` tag before the rollback deadline passes. The device commits the candidate configuration and cancels the rollback.

Contents

`<confirmed>`	Request a temporary commit of the candidate configuration. If the commit is not confirmed, the device reverts to the previous active configuration after a specified time, which is 600 seconds (10 minutes) by default.
`<confirm-timeout>`	Specify the number of seconds before the device reverts to the previously active configuration. If you omit this element, the server uses the default value. • **Range:** 1 through 4,294,967,295 seconds • **Default:** 600 seconds

RELATED DOCUMENTATION

Commit the Candidate Configuration Using NETCONF | 323

Commit the Candidate Configuration Only After Confirmation Using NETCONF | 325

<copy-config>

IN THIS SECTION

Usage

```
<rpc>
    <copy-config>
        <target>
            <url>
```

```
                <!-- location specifier for target file -->
            </url>
        </target>
        <source>
            <(candidate | running)/>
        </source>
    </copy-config>

    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
            <running/>
        </source>
    </copy-config>

    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
            <url format="(xml | text)">
                <!-- location specifier for input file -->
            </url>
        </source>
    </copy-config>
</rpc>
]]>]]>
```

Description

Copy the source configuration datastore to the target configuration datastore.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing a `<copy-config>` operation, `<candidate/>` refers to the open configuration database. Otherwise, `<candidate/>` refers to the candidate configuration.

You can use `<copy-config>` to perform the following operations:

- Copy the active or candidate configuration data in XML format to a file.
- Copy the active configuration to the candidate configuration or open configuration database.

- Copy the configuration data in the specified file to the candidate configuration or open configuration database.

Attributes

`format="(text | xml)"`	Specify the format of the source configuration data. • `text`—Load configuration data formatted as text. • `xml`—Load configuration data formatted as Junos XML tag elements. This is the default.

Contents

`<candidate/>`	Specify the open configuration database, or if there is no open database, the candidate configuration database.
`<running/>`	Specify the active configuration database.
`<source>`	Specify the source configuration datastore.
`<target>`	Specify the target configuration datastore.
`<url>`	Specify the location of the input or output file. For more information, see "Upload and Format Configuration Data in a NETCONF Session" on page 279.

Release Information

`<url>` (as target) support for `file://` URI with absolute path added in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1.

RELATED DOCUMENTATION

Replace the Candidate Configuration Using NETCONF | **293**

Upload and Format Configuration Data in a NETCONF Session | **279**

<delete-config>

IN THIS SECTION

Usage

```
<rpc>
    <delete-config>
        <target>
            <candidate/>
        </target>
    </delete-config>
</rpc>
]]>]]>
```

Description

Delete all configuration data in the existing candidate configuration or open configuration database.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<delete-config>` operation on the target `<candidate/>`, Junos OS performs the `<delete-config>` operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

Contents

The `<target>` tag element and its contents are explained separately.

RELATED DOCUMENTATION

Delete the Configuration Using NETCONF | 299

Delete Configuration Elements Using NETCONF | 306

<target> | 217

<discard-changes/>

IN THIS SECTION

Usage

```
<rpc>
    <discard-changes/>
</rpc>
]]>]]>
```

Description

Discard changes made to the candidate configuration and make its contents match the contents of the current running (active) configuration. This operation is equivalent to the Junos OS CLI configuration mode `rollback 0` command.

NOTE: The `<discard-changes/>` operation cannot be used to discard uncommitted changes that have been loaded into the ephemeral configuration database.

RELATED DOCUMENTATION

Roll Back Uncommitted Changes in the Candidate Configuration Using NETCONF | 298

<edit-config>

IN THIS SECTION

Usage

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>

    <!-- EITHER -->

        <config>
            <configuration>
                <!-- tag elements representing the data to incorporate -->
            </configuration>
        </config>

    <!-- OR -->

        <config-text>
            <configuration-text>
                <!-- configuration data in text format -->
            </configuration-text>
        </config-text>

    <!-- OR -->

        <url format="(xml | text)">
            <!-- location specifier for file containing data -->
        </url>
        <default-operation>(merge | none | replace)</default-operation>
        <error-option>(ignore-error | stop-on-error)</error-option>
        <test-option>(set | test-then-set)</test-option>
    </edit-config>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server incorporate configuration data into the candidate configuration or open configuration database. Provide the data in one of three ways:

- Data stream of Junos XML elements—Include the `<config>` tag element to provide a data stream of Junos XML configuration tag elements to incorporate. The tag elements are enclosed in the `<configuration>` tag element.
- Data stream of configuration data in text format—Include the `<config-text>` tag element to provide a data stream of CLI configuration statements to incorporate. The configuration statements are enclosed in the `<configuration-text>` tag element.
- File containing configuration data—Include the `<url>` tag element to specify the location of a file that contains the Junos OS configuration to incorporate. The format of the configuration data can be Junos XML elements or CLI configuration statements.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<edit-config>` operation on the target `<candidate/>`, Junos OS performs the `<edit-config>` operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

Contents

`<config>`	Enclose the `<configuration>` tag element.
`<configuration>`	Enclose configuration data in Junos XML format. This configuration data is provided as a data stream and is incorporated into the candidate configuration or open configuration database. For information about the syntax for representing the elements to create, delete, or modify, see "Map Configuration Statements to Junos XML Tag Elements" on page 25.
`<config-text>`	Enclose the `<configuration-text>` tag element.

`<configuration-text>`	Enclose configuration data formatted as CLI configuration statements. This configuration data is provided as a data stream and is incorporated into the candidate configuration or open configuration database.
`<default-operation>`	(Optional) Specify how to incorporate the new configuration data into the candidate configuration or open configuration database, particularly when there are conflicting statements. • **Default:** `merge` • **Values:** • `merge`—Combine the new configuration data with the existing configuration according to the rules defined in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. This mode applies to all elements in the new data that do not have the `operation` attribute in their opening container tag to specify a different mode. • `none`—Retain each configuration element in the existing configuration unless the new data includes a corresponding element that has the `operation` attribute in its opening container tag to specify an incorporation mode. This mode prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. See "Set the Edit Configuration Mode in a NETCONF Session" on page 287. • `replace`—Discard the existing configuration data in the candidate configuration or open configuration database and replace it with the new data. See "Replace the Candidate Configuration Using NETCONF" on page 293.
`<error-option>`	(Optional) Specify how the NETCONF server handles errors encountered while it incorporates the configuration data. • **Default:** `stop-on-error` • **Values:** • `ignore-error`—Instruct the NETCONF server to continue incorporating the new configuration data even if it encounters an error. • `stop-on-error`—Instruct the NETCONF server to stop incorporating the new configuration data when it encounters an error.
`<test-option>`	(Optional) Specify whether the NETCONF server validates the configuration data before incorporating it into the candidate configuration.

- **Default:** `test-then-set`
- **Values:**
  - `set`—Do not perform validation
  - `test-then-set`—Perform validation and do not incorporate the data if the validation fails

Regardless of the value provided, the NETCONF server performs a basic syntax check on the Junos OS configuration data in the `<edit-config>` tag element. It performs a complete syntactic and semantic validation on the candidate configuration in response to the `<validate>` and `<commit>` tag elements, but not for the `<edit-config>` tag element.

NOTE: The ephemeral configuration database does not support using the `<test-option>` element when loading configuration data into the database.

`<url>`

Specify the full pathname of the file that contains the configuration data to load.

When the configuration data is formatted as Junos XML tag elements, set the `<url>` `format` attribute to "xml" or omit the attribute. When the configuration data is formatted as CLI configuration statements, set the `<url>` format attribute to "text". For more information, see "Upload and Format Configuration Data in a NETCONF Session" on page 279.

The `<target>` tag element and its contents are explained separately.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.1R1 and 21.1R1-EVO	Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 21.1R1,when you set `<default-operation>` to `replace`, the device uses a `load update` operation instead of a `load override` operation to replace the configuration. In load update operations, the device notifies only the Junos processes that correspond to changed statements, thus minimizing possible disruptions to the network.

RELATED DOCUMENTATION

<get>

IN THIS SECTION

Usage

```
<rpc>
    <get [format="(json | json-minified | set | text | xml | xml-minified)"]>
        <filter type="subtree">
            <configuration>
                <!-- tag elements representing the configuration elements to return -->
            </configuration>
        </filter>
    </get>

    <get>
        <filter type="subtree">
            <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
                (<capabilities/> | <datastores/> | <schemas/> | <sessions/> | <statistics/>)
            </netconf-state>
        </filter>
    </get>

</rpc>
]]>]]>
```

Description

Request the committed configuration or NETCONF state information from the NETCONF server.

Attributes

`format` Specify the return format for the configuration data.

- **Default:** `xml`
- **Values:**
  - `json`—Configuration statements are formatted in JSON.
  - `json-minified`—Configuration statements are formatted in JSON with unnecessary spaces, tabs, and newlines removed.
  - `set`—Configuration statements are formatted as Junos OS configuration mode commands.
  - `text`—Configuration statements are formatted as ASCII text, using the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements.
  - `xml`—Configuration statements are represented by the corresponding Junos XML tag elements.
  - `xml-minified`—Configuration statements are represented by the corresponding Junos XML tag elements with unnecessary spaces, tabs, and newlines removed.

Contents

`<filter>` (Optional) Specify the information that the NETCONF server should return.

- **Values:**

- `<configuration>`—Return the committed configuration.

  To specify the configuration elements to return, optionally include the Junos XML tag elements that represent all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to each element to display.

- `<netconf-state>`—Return NETCONF state data for the requested subtree. You must specify one of the following:

  - `capabilities`—NETCONF capabilities supported by the NETCONF server.
  - `datastores`—Available configuration datastores, for example, candidate or running (active), and their lock state.
  - `schemas`—Schemas supported on the device.
  - `sessions`—Active NETCONF management sessions on the device.
  - `statistics`—NETCONF server performance data.

NOTE: The optional `type` attribute indicates the kind of syntax used to represent the requested configuration elements or state information; the only acceptable value is `subtree`.

Release Information

`<netconf-state>` `<schemas>` added in Junos OS Release 21.1R1 and Junos OS Evolved Release 21.1R1.

`<netconf-state>` `<capabilities>`, `<datastores>`, `<sessions>`, and `<statistics>` added in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1.

RELATED DOCUMENTATION

Request the Committed Configuration and Device State Using NETCONF | 398

<get-config>

IN THIS SECTION

Usage

```
<rpc>
    <get-config>
        <source>
            <( candidate | running )/>
        </source>
    </get-config>

    <get-config>
        <source>
            <( candidate | running )/>
        </source>
        <filter type="subtree">
            <configuration>
                <!-- tag elements for each configuration element to return -->
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
```

Description

Request configuration data from the NETCONF server. The child tag elements `<source>` and `<filter>` specify the source and scope of data to display:

- To display the entire active configuration, enclose the `<source>` tag element and `<running/>` tag in the `<get-config>` tag element.
- To display either the entire candidate configuration or all configuration data in the open configuration database, enclose the `<source>` tag element and `<candidate/>` tag in the `<get-config>` tag element.

  If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-config>` operation, setting the source to `<candidate/>` retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.
- To display one or more sections of the configuration hierarchy (hierarchy levels or configuration objects), enclose the appropriate child tag elements in the `<source>` and `<filter>` tag elements.

Contents

`<candidate/>`	Specify the open configuration database, or if there is no open database, the candidate configuration.
`<configuration>`	Enclose tag elements that specify which configuration elements to return.
`<filter>`	Enclose the `<configuration>` tag element. The mandatory `type` attribute indicates the kind of syntax used to represent the requested configuration elements; the only acceptable value is `subtree`. To specify the configuration elements to return, include within the `<filter>` tag element the Junos XML tag elements that represent all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to each element to display. For information about the configuration elements available in the current version of the Junos OS, see the XML API Explorer.
`<running/>`	Specify the active (mostly recently committed) configuration.
`<source>`	Enclose the tag that specifies the source of the configuration data. To specify either the candidate configuration or an open configuration database, include the `<candidate/>` tag. To specify the active configuration, include the `<running/>` tag.

Usage Guidelines

See "Request Configuration Data Using NETCONF" on page 400.

RELATED DOCUMENTATION

<data> | 209

<kill-session>

IN THIS SECTION

Usage

```
<rpc>
    <kill-session>
        <session-id>PID</session-id>
    </kill-session>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server terminate another CLI or NETCONF session. The usual reason to emit this tag is that the user or application for the other session holds a lock on the candidate configuration, preventing the client application from locking the configuration itself.

The client application must have the Junos OS `maintenance` permission to perform this operation.

Contents

`<session-id>`	Process identifier (PID) of the entity conducting the session to terminate. The PID is reported in the `<rpc-error>` tag element that the NETCONF server generates when it cannot lock a configuration as requested.

NOTE: Starting in Junos OS Release 19.1R1, if the session identifier is equal to the current session ID, the values of the `<error-type>` and `<error-tag>` elements in the resulting `<rpc-error>` are `application` and `invalid-value`, respectively. In earlier releases, the `<error-type>` and `<error-tag>` values are `protocol` and `operation-failed`.

RELATED DOCUMENTATION

Terminate a NETCONF Session | **134**

<lock>

IN THIS SECTION

Usage

```
<rpc>
    <lock>
        <target>
            <candidate/>
        </target>
    </lock>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server lock the candidate configuration, enabling the client application both to read and change it, but preventing any other users or applications from changing it. The client application must emit the `<unlock/>` tag to unlock the configuration.

If the NETCONF session ends or the application emits the `<unlock>` tag element before the candidate configuration is committed, all changes made to the candidate are discarded.

Contents

The `<target>` tag element and its contents are explained separately.

RELATED DOCUMENTATION

Lock and Unlock the Candidate Configuration | 132

<target> | 217

<unlock> | 204

operation

IN THIS SECTION

Usage

```
<rpc>
  <edit-config>
    <config>
      <configuration>
        <!-- opening tags for each parent of the changing element -->
```

```
          <changing-element operation="(create | delete | remove | replace)">
            <name>identifier</name>
            <!-- if changing element has an identifier -->
            <!-- other child tag elements, if appropriate -->
          </changing-element>
        <!-- closing tags for each parent of the changing element -->
      </configuration>
    </config>
    <!-- other child tag elements of the <edit-config> tag element -->
  <edit-config>
</rpc>
]]>]]>
```

Description

Specify how the NETCONF server incorporates an individual configuration element into the target configuration, which can be either the candidate configuration or the open configuration database. If you omit the attribute, the element is merged into the configuration according to the rules defined in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. The NETCONF server accepts the following values:

`create`	Create the specified element in the target configuration only if the element does not already exist.
`delete`	Delete the specified element from the target configuration. If the element does not exist in the target configuration, the server returns an `<rpc-error>` element with an `<error-tag>` value of `data-missing`. When you use this attribute, we recommend that you include the `<default-operation>none</default-operation>` element in the `<edit-config>` RPC.
`remove`	Delete the specified element from the target configuration. If the element does not exist in the target configuration, the server silently ignores the request and does not return any error related to this request. When you use this attribute, we recommend that you include the `<default-operation>none</default-operation>` element in the `<edit-config>` RPC.
`replace`	Replace the specified element in the target configuration with new configuration data.

NOTE: The `operation="replace"` attribute is not supported when loading configuration data into the ephemeral configuration database.

Release Information

Starting in Junos OS Release 23.1R1 and Junos OS Evolved Release 23.1R1, the NETCONF server `<rpc-error>` response is changed when `<edit-config>` uses the `operation="delete"` operation to delete a configuration element that is absent in the target configuration. The error severity is error instead of warning, and the `<rpc-error>` element includes the `<error-tag>data-missing</error-tag>` and `<error-type>application</error-type>` elements.

RELATED DOCUMENTATION

Change Individual Configuration Elements Using NETCONF | **300**

Create Configuration Elements Using NETCONF | **304**

Delete Configuration Elements Using NETCONF | **306**

Replace Configuration Elements Using NETCONF | **314**

Set the Edit Configuration Mode in a NETCONF Session | **287**

<edit-config> | **191**

<unlock>

IN THIS SECTION

Usage

```
<rpc>
    <unlock>
        <target>
            <candidate/>
        </target>
    </unlock>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server unlock and close the candidate configuration, which the client application previously locked by emitting the `<lock>` tag element. Until the application emits this tag element, other users or applications can read the configuration but cannot change it.

Contents

The `<target>` tag element and its contents are explained separately.

RELATED DOCUMENTATION

Lock and Unlock the Candidate Configuration | **132**

<lock> | **201**

<target> | **217**

<validate>

IN THIS SECTION

Usage

```
<rpc>
    <validate>
        <source>
            <candidate/>
        </source>
    </validate>
</rpc>
]]>]]>
```

Description

Check that the candidate configuration is syntactically valid.

Contents

`<source>`	Enclose the tag that specifies the configuration to validate.
`<candidate/>`	Specify the candidate configuration.

Release Information

Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, when you configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level, the NETCONF server emits only an `<ok/>` or `<rpc-error>` element in response to `<validate>` operations. In earlier releases, the RPC reply also includes the `<commit-results>` element.

RELATED DOCUMENTATION

Verify the Candidate Configuration Syntax Using NETCONF | 322

CHAPTER 7

NETCONF Request and Response Tags

IN THIS CHAPTER

End-of-document Character Sequence

IN THIS SECTION

Usage

```
<hello>
    <!-- child tag elements included by client application or NETCONF server -->
```

```
</hello>
]]>]]>
```

```
<rpc [attributes]>
    <!-- tag elements in a request from a client application -->
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <!-- tag elements in the response from the NETCONF server -->
</rpc-reply>
]]>]]>
```

Description

Signal the end of each XML document sent by the NETCONF server and client applications. A client application sends the sequence after its closing `</hello>` tag and each closing `</rpc>` tag. The NETCONF server sends the sequence after its closing `</hello>` tag and each closing `</rpc-reply>` tag.

Use of this signal is required by RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, available at http://www.ietf.org/rfc/rfc4742.txt .

NOTE: Starting in Junos OS Release 24.4R1 and Junos OS Evolved Release 24.4R1, you can configure the NETCONF session to comply with RFC 6242, *Using the NETCONF Protocol over Secure Shell (SSH).* If you enable RFC 6242 compliance and both peers advertise the `:base:1.1` capability, the NETCONF session uses chunked framing for all messages after the initial `<hello>` message. Otherwise, the NETCONF session uses the end-of-document character sequence (]]>]]>) as the message separator.

RELATED DOCUMENTATION

Generate Well-Formed XML Documents | 37

<hello> | 211

<rpc> | 213

<rpc-reply> | 216

<data>

IN THIS SECTION

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration>
            <!-- Junos XML tag elements for the configuration data -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

Description

Encloses configuration data and device information returned by the NETCONF server in response to a `<get>` request or configuration data returned by the NETCONF server in response to a `<get-config>` request.

NOTE: The NETCONF server, by default, returns configuration data formatted as Junos XML tag elements. The configuration data enclosed in the `<data>` element can vary if a client application requests a different format in a `<get>` request.

Contents

`<configuration>`	Encloses configuration tag elements. It is the top-level tag element in the Junos XML API, equivalent to the `[edit]` hierarchy level in the Junos OS CLI. For information about

Junos OS configuration elements, see the *Junos XML API Configuration Developer Reference*.

Usage Guidelines

See "Request Configuration Data Using NETCONF" on page 400.

RELATED DOCUMENTATION

<error-info>

IN THIS SECTION

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rpc-error>
        <error-info>
            <bad-element>command-or-statement</bad-element>
        </error-info>
    </rpc-error>
</rpc-reply>
]]>]]>
```

Description

Provides additional information about the event or condition that causes the NETCONF server to report an error or warning in the `<rpc-error>` tag element.

Contents

`<bad-element>`	Identifies the command or configuration statement that was being processed when the error or warning occurred. For a configuration statement, the `<error-path>` tag element enclosed in the `<rpc-error>` tag element specifies the statement's parent hierarchy level.

RELATED DOCUMENTATION

Handle an Error or Warning in a NETCONF Session | **130**

<rpc-error> | **214**

<rpc-reply> | **216**

<hello>

IN THIS SECTION

Usage

```
<!-- emitted by a client application -->
<hello>
    <capabilities>
        <capability>URI</capability>
    </capabilities>
```

```
</hello>
]]>]]>
```

```
<!-- emitted by the NETCONF server -->
<hello>
    <capabilities>
        <capability>URI</capability>
    </capabilities>
    <session-id>session-identifier</session-id>
</hello>
]]>]]>
```

Description

Specify which operations, or *capabilities*, the emitter supports from among those defined in the NETCONF specification. The client application must emit the `<hello>` tag element before any other tag element during the NETCONF session, and must not emit it more than once.

Contents

`<capabilities>`	Encloses one or more `<capability>` tags, which together specify the set of supported NETCONF operations.
`<capability>`	Specifies the uniform resource identifier (URI) of a capability defined in the NETCONF specification or by a vendor. Each capability from the NETCONF specification is represented by a uniform resource name (URN). Capabilities defined by vendors are represented by URNs or URLs.
`<session-id>`	(Generated by NETCONF server only) Specifies the UNIX process ID (PID) of the NETCONF server for the session.

RELATED DOCUMENTATION

Exchanging <hello> Tag Elements | **119**

<ok/>

IN THIS SECTION

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

Description

Indicates that the NETCONF server successfully performed the requested operation that changes the state or contents of the device configuration.

RELATED DOCUMENTATION

<rpc-reply> | 216

<rpc>

IN THIS SECTION

Usage

```
<rpc [attributes]>]
    <!-- tag elements in a request from a client application -->
</rpc>
]]>]]>
```

Description

Enclose all tag elements in a request generated by a client application.

Attributes

(Optional) One or more attributes of the form *attribute-name*="*value*". This feature can be used to associate requests and responses if the value assigned to an attribute by the client application is unique in each opening `<rpc>` tag. The NETCONF server echoes the attribute unchanged in its opening `<rpc-reply>` tag, making it simple to map the response to the initiating request. The NETCONF specification assigns the name `message-id` to this attribute.

RELATED DOCUMENTATION

Send Requests to the NETCONF Server | 124

<rpc-reply> | 216

<rpc-error>

IN THIS SECTION

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rpc-error>
        <error-severity>error-severity</error-severity>
        <error-path>error-path</error-path>
        <error-message>error-message</error-message>
        <error-info>...</error-info>
    </rpc-error>
</rpc-reply>
]]>]]>
```

Description

Indicate that the NETCONF server has experienced an error while processing the client application's request. If the server has already emitted the response tag element for the current request, the information enclosed in that response tag element might be incomplete. The client application must include code that discards or retains the information, as appropriate. The child tag elements described in the Contents section detail the nature of the error. The NETCONF server does not necessarily emit all child tag elements; it omits tag elements that are not relevant to the current request.

NOTE: Starting in Junos OS Release 17.4R3, 18.2R2, 18.3R2, and 18.4R1, when you configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level to enforce certain behaviors by the NETCONF server, the NETCONF server cannot return an RPC reply that includes both an `<rpc-error>` element and an `<ok/>` element. If the operation is successful, but the server reply would include one or more `<rpc-error>` elements with a severity level of warning in addition to the `<ok/>` element, then the warnings are omitted.

Contents

`<error-message>`	Describes the error or warning in a natural-language text string.
`<error-path>`	Specifies the path to the Junos OS configuration hierarchy level at which the error or warning occurred, in the form of the CLI configuration mode banner.
`<error-severity>`	Indicates the severity of the event that caused the NETCONF server to return the `<rpc-error>` tag element. The two possible values are `error` and `warning`.

The `<error-info>` tag element is described separately.

RELATED DOCUMENTATION

<rpc-reply>

IN THIS SECTION

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <!-- tag elements in a reply from the NETCONF server-->
</rpc-reply>
]]>]]>
```

Description

Encloses all tag elements in a reply from the NETCONF server. The immediate child tag element is usually one of the following:

- Junos XML response tag—Encloses the data that the client application requests using a Junos XML operational request tag. For example, the server returns the `<interface-information>` tag in response to the `<get-interface-information>` request tag.
- `<data>`—Encloses the data that the client application requests using either the `<get>` or the `<get-config>` tag element.
- `<ok/>`—Confirms that the NETCONF server successfully performed an operation that changes the state or contents of the configuration (such as a lock, change, or commit operation)

- `<output>`—Encloses data for requests where the Junos XML API does not define a specific tag element for the requested operational information.
- `<rpc-error>`—Encloses errors or warnings.

Attributes

`xmlns`	Name of the default XML namespace for the enclosed tag elements.

RELATED DOCUMENTATION

Parse the NETCONF Server Response | **127**

<data> | **209**

<ok/> | **213**

<rpc> | **213**

<rpc-error> | **214**

<target>

IN THIS SECTION

Usage

```
<rpc>
    <( copy-config | delete-config | edit-config | lock | unlock )>
        <target>
            <candidate/>
        </target>
        ...
    </( copy-config | delete-config | edit-config | lock | unlock )>

    <copy-config>
        <target>
            <url>
                <!-- location specifier for target file -->
            </url>
        </target>
        <source>
            <(candidate | running)/>
        </source>
    </copy-config>
</rpc>
]]>]]>
```

Description

Specify the target configuration datastore for the specified operation.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing a `<copy-config>`, `<delete-config>`, or `<edit-config>` operation on the target `<candidate/>`, the device performs the requested operation on the open configuration database. Otherwise, the device performs the operation on the candidate configuration. Client applications can only perform the `<lock>` and `<unlock>` operations on the candidate configuration.

Contents

`<candidate/>`	Specify the target configuration database for the specified operation, either the open configuration database, or if there is no open database, the candidate configuration. This is the only acceptable value for Junos devices.
`<url>`	For `<copy-config>` operations, specify the output file location.

RELATED DOCUMENTATION

Delete the Configuration Using NETCONF | **299**

Edit the Configuration Using NETCONF | **277**

Replace the Candidate Configuration Using NETCONF | **293**

Lock and Unlock the Candidate Configuration | **132**

CHAPTER 8

Junos XML Protocol Elements Supported in NETCONF Sessions

IN THIS CHAPTER

<abort/>

IN THIS SECTION

Usage

```
<rpc>
    <!-- child tag elements -->
</rpc>
<abort/>
```

Description

Direct the NETCONF or Junos XML protocol server to stop processing the request that is currently outstanding. The server responds by returning the `<abort-acknowledgment/>` tag. If the server already sent tagged data in response to the request, the client application must discard those elements.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Halt a Request in Junos XML Protocol Sessions

<abort-acknowledgement/>

<abort-acknowledgement/>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <any-child-of-rpc-reply>
        <abort-acknowledgement/>
    </any-child-of-rpc-reply>
</rpc-reply>
```

Description

Indicates that the NETCONF or Junos XML protocol server has received the `<abort/>` tag and has stopped processing the current request. If the client application receives any tag elements related to the request between sending the `<abort/>` tag and receiving this tag, it must discard them.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<abort/>

<checksum-information>

IN THIS SECTION

Usage

```
<rpc-reply>
    <checksum-information>
        <file-checksum>
            <computation-method>MD5</computation-method>
            <input-file>
                <!-- name and path of file-->
            </input-file>
        </file-checksum>
    </checksum-information>
</rpc-reply>
```

Description

Encloses tag elements that include the file to check, the checksum algorithm used, and the checksum output.

Contents

`<checksum>`	Resulting value from the checksum computation.
`<computation-method>`	Checksum algorithm used. Currently, all checksum computations use the MD5 algorithm; thus, the only possible value is MD5.

`<file-checksum>`	Wrapper that holds the resulting `<input-file>`, `<computation-method>`, and `<checksum>` attributes for a particular checksum computation.
`<input-file>`	Name and path of the file that the checksum algorithm was run against.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<get-checksum-information>

<close-configuration/>

IN THIS SECTION

Usage

```
<rpc>
    <close-configuration/>
</rpc>
```

Description

Close the open configuration database and discard any uncommitted changes.

This operation is normally used to close a private copy of the candidate configuration or an open instance of the ephemeral configuration database and discard any uncommitted changes. The application must have previously emitted the `<open-configuration>` operation. Closing the NETCONF or Junos XML protocol session (by emitting the `<request-end-session/>` tag, for example) has the same effect as emitting this operation.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Lock, Unlock, or Create a Private Copy of the Candidate Configuration Using the Junos XML Protocol

<open-configuration>

<request-end-session/>

<commit-configuration>

IN THIS SECTION

Usage

```
<rpc>
    <commit-configuration/>

    <commit-configuration>
        <check/>
```

```
</commit-configuration>

<commit-configuration>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <at-time>time-specification</at-time>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <synchronize/>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <synchronize/>
    <at-time>time-specification</at-time>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <synchronize/>
    <check/>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <synchronize/>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
    <log>log-message</log>
</commit-configuration>

<commit-configuration>
    <synchronize/>
```

```
        <force-synchronize/>
    </commit-configuration>
</rpc>
```

Description

Request that the NETCONF or Junos XML protocol server perform one of the variants of the commit operation. You can perform the commit operation on the candidate configuration, a private copy of the candidate configuration, or an open instance of the ephemeral configuration database.

On devices with dual Routing Engines, you can commit the candidate configuration, private copy, or ephemeral database instance stored on the local Routing Engine on both Routing Engines. The ephemeral database supports only the `<synchronize/>` option.

Some restrictions apply to the commit operation for a private copy of the candidate configuration and for the ephemeral configuration database. For example:

- The commit operation fails for a private copy if the regular candidate configuration is locked by another user or application or if it includes uncommitted changes made since the private copy was created.
- A commit operation on an instance of the ephemeral configuration database supports only the `<synchronize/>` option.
- The confirmed commit operation is not available when committing a private copy of the configuration or an open instance of the ephemeral configuration database.

To execute a commit or commit synchronize operation, enclose the appropriate tags in the `<commit-configuration>` tag element to specify the type of commit operation. Table 12 on page 226 and Table 13 on page 227 describe common commit and commit synchronize operations.

Table 12: Commit Operations

`<commit-configuration>` Operation	Description
`<commit-configuration/>`	Commit the configuration immediately, making it the active configuration on the device.

`<commit-configuration>`
    `<check/>`
`</commit-configuration>`
	Verify the syntactic correctness of the candidate configuration or a private copy without actually committing it.

Table 12: Commit Operations *(Continued)*

`<commit-configuration>` Operation	Description

`<commit-configuration>`
`    <confirmed/>`
`</commit-configuration>`
 	
`<commit-configuration>`
`    <confirmed/>`
`    <confirm-timeout>`*`rollback-delay`*`</`
`confirm-timeout>`
`</commit-configuration>`
	Commit the candidate configuration but require an explicit confirmation for the commit to become permanent. If the commit is not confirmed, the configuration rolls back to the previous configuration after the specified time.
	
	Optionally include the `<confirm-timeout>` element to specify the rollback delay in the range from 1 through 65,535 minutes. By default, the rollback occurs after 10 minutes.
	
	To delay the rollback again (past the original rollback deadline), emit the `<commit-configuration><confirmed/></commit-configuration>` tags before the deadline passes, and optionally Include the `<confirm-timeout>` element. The rollback can be delayed repeatedly in this way.
	
	To confirm the commit, emit the empty `<commit-configuration/>` tag or the `<commit-configuration><check/><commit-configuration>` tags before the rollback deadline passes. The device commits the candidate configuration and cancels the rollback.

Table 13: Commit Synchronize Operations

`<commit-configuration>` Operation	Description

`<commit-configuration>`
`    <synchronize/>`
`</commit-configuration>`
	Copy the candidate configuration or the open ephemeral instance data from the local Routing Engine to the other Routing Engine, verify the configuration's syntactic correctness, and commit it immediately on both Routing Engines.

`<commit-configuration>`
`    <synchronize/>`
`    <at-time>`*`time-specification`*`</at-time>`
`</commit-configuration>`
	Copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines at a defined future time.
	
	You can also specify `<force-synchronize/>`.

Table 13: Commit Synchronize Operations *(Continued)*

`<commit-configuration>` Operation	Description

`<commit-configuration>`
    `<synchronize/>`
    `<check/>`
`</commit-configuration>`
	Copy the candidate configuration stored on the local Routing Engine to the other Routing Engine and verify the candidate's syntactic correctness on each Routing Engine.
	
	You can also specify `<force-synchronize/>`.

`<commit-configuration>`
    `<synchronize/>`
    `<confirmed/>`
    `<confirm-timeout>`*`rollback-delay`*`</confirm-timeout>`
`</commit-configuration>`
	Copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines but require confirmation.

`<commit-configuration>`
    `<synchronize/>`
    `<force-synchronize/>`
`</commit-configuration>`
	Force the same synchronized commit operation as invoked by the `<synchronize/>` tag to succeed, even if there are open configuration sessions or uncommitted configuration changes on the remote machine.

To schedule the candidate configuration for commit at a future time, enclose the `<at-time>` element in the `<commit-configuration>` element. When you execute the operation, the configuration is checked immediately for syntactic correctness. If the check succeeds, the configuration is scheduled for commit at the specified time. If the check fails, the commit operation is not scheduled. Table 14 on page 228 outlines the valid types of time specifiers.

Table 14: <at-time> Time Specifiers

Time Specifier	Description	Example
`reboot`	Commit the configuration the next time the device reboots.	• `<at-time>reboot</at-time>`

Table 14: <at-time> Time Specifiers *(Continued)*

Time Specifier	Description	Example
hh:*mm*[:*ss*]	Commit the configuration at the specified time (hours, minutes, and, optionally, seconds). The time must be in the future but before 11:59:59 PM on the current day. Use 24-hour time for the *hh* value. The device interprets the time with respect to its clock and time zone settings.	• Execute the operation at 4:30:00 AM: `<at-time>04:30:00</at-time>` • Execute the operation at 8:00 PM: `<at-time>20:00</at-time>`
yyyy-*mm*-*dd* *hh*:*mm*[:*ss*]	Commit the configuration at the specified date and time (year, month, date, hours, minutes, and, optionally, seconds). The specified time must be after you execute the `<commit-configuration>` operation. Use 24-hour time for the *hh* value. The device interprets the time with respect to its clock and time zone settings.	• Execute the operation at 3:30 PM on August 21, 2005: `<at-time>2005-08-21 15:30:00</at-time>`

Contents

`<at-time>`	Schedule the commit operation for a specified future time. Valid time specifiers include: • `reboot` • *hh*:*mm*[:*ss*] • *yyyy*-*mm*-*dd* *hh*:*mm*[:*ss*]
`<check>`	Request verification that the configuration is syntactically correct, but do not actually commit it.
`<confirmed>`	Request a commit of the candidate configuration and require an explicit confirmation for the commit to become permanent. If the commit is not confirmed, roll back to the previous configuration after a short time, 10 minutes by default. Use the `<confirm-timeout>` tag element to specify a different amount of time.
`<confirm-timeout>`	Specify the number of minutes for which the configuration remains active when the `<confirmed/>` tag is enclosed in the `<commit-configuration>` tag element.

- **Range:** 1 through 65,535 minutes
- **Default:** 10 minutes

`<log>`	Record a message in the commit history log when the commit operation succeeds.
`<synchronize>`	On dual control plane systems, request that the configuration on one control plane be copied to the other control plane, checked for correct syntax, and committed on both Routing Engines.
`<force-synchronize>`	On dual control plane systems, force the candidate configuration on one control plane to be copied to the other control plane.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Commit the Candidate Configuration Using the Junos XML Protocol

Commit a Private Copy of the Configuration Using the Junos XML Protocol

Committing a Configuration at a Specified Time Using the Junos XML Protocol

Commit the Candidate Configuration Only After Confirmation Using the Junos XML Protocol

Commit and Synchronize a Configuration on Redundant Control Planes Using the Junos XML Protocol

<commit-results>

<commit-results>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <!-- for the candidate configuration or ephemeral configuration -->
    <commit-results>
        <routing-engine>...</routing-engine>
    </commit-results>

    <!-- for a private copy -->
    <commit-results>
        <load-success/>
        <routing-engine>...</routing-engine>
    </commit-results>

    <!-- for a private copy that does not include changes -->
    <commit-results>
    </commit-results>

</rpc-reply>
```

Description

Tag element returned by the Junos XML protocol server in response to a `<commit-configuration>` request by a client application. The `<commit-results>` element contains information about the requested commit operation performed by the server on a particular Routing Engine.

Contents

`<load-success/>`	Indicates that the Junos XML protocol server successfully merged changes from the private copy into a copy of the candidate configuration, before committing the combined candidate on the specified Routing Engine.

The `<routing-engine>` tag element is described separately.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Committing the Candidate Configuration Using the Junos XML Protocol

<commit-configuration>

<routing-engine>

<commit-revision-information>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <commit-results>
        <routing-engine>

            <!-- configuration with commit revision identifier -->
            <commit-revision-information>
                <old-db-revision>old-revision-id</old-db-revision>
                <new-db-revision>new-revision-id</new-db-revision>
            </commit-revision-information>

        </routing-engine>
    </commit-results>

</rpc-reply>
```

Description

Child element included in a Junos XML protocol server `<commit-results>` response element to return information about the old and new configuration revision identifiers (CRI) on a particular Routing Engine. The CRI is a unique string (for example, re0-1365168149-1) that is associated with a committed configuration.

Network management system (NMS) applications, such as Junos Space, can use the configuration revision identifier to determine if the NMS's known configuration for a Junos device is identical to the device's current configuration. The NMS can detect if out-of-band commits were made to the device by comparing the CRI associated with the NMS's last commit to the CRI of the configuration on the device.

Contents

`<old-db-revision>`	Indicates the old configuration revision identifier, which is the identifier of the configuration prior to the previously successfully committed configuration.
`<new-db-revision>`	Indicates the new configuration revision identifier, which is the identifier of the last successfully committed configuration.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

Element introduced in Junos OS Release 16.1.

RELATED DOCUMENTATION

View the Configuration Revision Identifier for Determining Synchronization Status of Devices with NMS

<commit-results>

<routing-engine>

<database-status>

IN THIS SECTION

Usage

```
<xnm:error>
    <database-status-information>
        <database-status>
            <user>username</user>
            <terminal>terminal</terminal>
            <pid>pid</pid>
            <start-time>start-time</start-time>
            <idle-time>idle-time</idle-time>
            <commit-at>time</commit-at>
            <exclusive/>
            <edit-path>edit-path</edit-path>
        </database-status>
    </database-status-information>
</xnm:error>
```

Description

The `database-status` element describes a user or NETCONF client application that is logged in to the configuration database. For simplicity, we use the term user to refer to both human users and client applications, except where the information differs for the two.

Contents

`<commit-at/>`	Indicate that the user has scheduled a commit operation for a later time.

`<edit-path>`	Specify the user's current location in the configuration hierarchy, in the form of the CLI configuration mode banner.
`<exclusive/>`	Indicate that the user or application has an exclusive lock on the configuration database. A user enters exclusive configuration mode by issuing the `configure exclusive` command in CLI operational mode. A client application obtains the lock by emitting the `<lock-configuration/>` tag element.
`<idle-time>`	Specify how much time has passed since the user last performed an operation in the database.
`<pid>`	Specify the process ID of the Junos OS management process (mgd) that is handling the user's login session.
`<start-time>`	Specify the time when the user logged in to the configuration database, in the format *YYYY-MM-DD hh:mm:ss TZ* (year, month, date, hour in 24-hour format, minute, second, time zone).
`<terminal>`	Identify the UNIX terminal assigned to the user's connection.
`<user>`	Specify the Junos OS login ID of the user whose login to the configuration database caused the error.

Release Information

This is a Junos XML management protocol response tag. It is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<database-status-information> | 236
<xnm:error> | 261

<database-status-information>

IN THIS SECTION

Usage

```
<data>
    <database-status-information>
        <database-status>...</database-status>
    </database-status-information>
</data>
```

```
<xnm:error>
    <database-status-information>
        <database-status>...</database-status>
    </database-status-information>
</xnm:error>
```

Description

Describes one or more users who have an open editing session in the configuration database.

The `<database-status>` tag element is explained separately.

Release Information

This is a Junos XML management protocol response tag. It is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<database-status> | 234

<xnm:error> | 261

<end-session/>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <end-session/>
</rpc-reply>
```

Description

Indicates that the NETCONF or Junos XML protocol server is about to end the current session for a reason other than an error. Most often, the reason is that the client application has sent the `<request-end-session/>` tag.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Ending a Junos XML Protocol Session and Closing the Connection

<request-end-session/>

<get-checksum-information>

IN THIS SECTION

Usage

```
<rpc>
    <get-checksum-information>
        <path>
        <!-- name and path of file -->
        </path>
    </get-checksum-information>
</rpc>
```

Description

Request checksum information for the specified file.

Contents

`<path>`	Name and path of the file to check.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

Operation added in Junos OS Release 9.2R1.

RELATED DOCUMENTATION

<checksum-information>

<get-configuration>

IN THIS SECTION

Usage

```
<rpc>
    <get-configuration
        [changed="changed"]
        [commit-scripts="( apply | apply-no-transients | view )"]
        [compare=("configuration-revision" [configuration-revision="revision-id"] | "rollback"
[rollback="[0-49]"])]
        [database="(candidate | committed)"]
        [database-path=$junos-context/commit-context/database-path]
        [format="( json | set | text | xml )"]
        [inherit="( defaults | inherit )"
            [groups="groups"] [interface-ranges="interface-ranges"]]
        [(junos:key | key )="key"] >
        <!-- tag elements for the configuration element to display -->
    </get-configuration>
</rpc>
```

Description

Request configuration data from the NETCONF or Junos XML protocol server. The attributes specify the source and formatting of the data to display.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-configuration>` operation, the server returns the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration. You can explicitly request the active configuration database by including the `database="committed"` attribute.

A client application can request the entire configuration hierarchy or a subset of it.

- Entire configuration hierarchy—To display the entire configuration hierarchy, emit the empty `<get-configuration/>` tag.

- Subset of configuration hierarchy—To display a configuration element (hierarchy level or configuration object), emit the `<get-configuration>` element and include the elements that represent all levels of the configuration hierarchy from the root (`<configuration>`) down to the level or object to display. To represent a hierarchy level or a configuration object that does not have an identifier, emit it as an empty tag. To represent an object that has one or more identifiers, emit its container tag element and identifier tag elements only, not any tag elements that represent other characteristics.

NOTE: To retrieve configuration data from an instance of the ephemeral configuration database, a client application must first open the ephemeral instance using the `<open-configuration>` operation with the appropriate child tags before emitting the `<get-configuration>` operation. When retrieving ephemeral configuration data using the `<get-configuration>` operation, the only supported attributes are `format` and `key`.

NOTE: You can use the `<get-configuration>` operation to request the entire logical system configuration or request specific logical system configuration hierarchies using child configuration tags.

Attributes

`changed` Request that the `junos:changed="changed"` attribute appear in the opening tag of each changed configuration element.

The attribute appears in the opening tag of every parent tag in the path to the changed configuration element, including the top-level `<configuration>` tag. If the changed configuration element is represented by a single (empty) tag, the `junos:changed="changed"` attribute appears in the tag. If the changed element is represented by a container tag, the `junos:changed="changed"` attribute appears in the opening container tag and also in each child tag enclosed in the container tag element.

- **Values:** `changed`

The `database` attribute can be combined with the `changed="changed"` attribute to request either the candidate or active configuration:

- When you request the candidate configuration, elements added to the candidate configuration after the last commit operation are marked with the `junos:changed="changed"` attribute.
- When you request the active configuration, elements added to the active configuration by the most recent commit are marked with the `junos:changed="changed"` attribute.

NOTE: When a commit operation succeeds, the server removes the `junos:changed="changed"` attribute from all elements. However, if warnings are generated during the commit, the attribute is not removed. In this case, the `junos:changed="changed"` attribute appears in elements that changed before the commit operation as well as on those elements that changed after it. To remove the `junos:changed="changed"` attribute from elements that changed before the commit, you must eliminate the cause of the warning, and commit the configuration again.

`commit-scripts` Request that the NETCONF or Junos XML protocol server display commit-script-style XML data. The value of the attribute determines the output.

- **Values:**
  - `apply`—Display the configuration with commit script changes applied, including both transient and non-transient changes. The output is equivalent to the `| display commit-scripts` output in the CLI.

- `apply-no-transients`—Display the configuration with commit script changes applied, but exclude transient changes. The output is equivalent to the `| display commit-scripts no-transients` output in the CLI.

- `view`—Display the configuration in the XML format that is input to a commit script. The output is equivalent to viewing the configuration with the following attributes applied: `inherit="inherit"`, `groups="groups"`, and `changed="changed"`. The output is equivalent to the `| display commit-scripts view` output in the CLI.

`compare` Request that the NETCONF or Junos XML protocol server display the differences between the active or candidate configuration and a previously committed configuration (the comparison configuration). By default, the comparison uses the candidate configuration. Include the `database` attribute to specify the active configuration.

- **Values:**

  - `configuration-revision`—Reference the comparison configuration by its configuration revision ID string, which you define in the `configuration-revision="`*`revision-id`*`"` attribute.

  - `rollback`—Reference the comparison configuration by its rollback index, which you define in the `rollback="`*`rollback-number`*`"` attribute.

If you include the `compare` attribute but either omit the corresponding `configuration-revision` or `rollback` attribute or provide an invalid configuration revision ID, the server uses the most recently committed configuration as the comparison configuration.

When you compare the candidate configuration to the active configuration, the `compare` operation returns XML output. However, you can include the `format` attribute to display the differences in text, XML, or JSON format. For all other comparisons, the server returns the output as text using a patch format.

NOTE: When you compare the candidate and active configurations and display the differences in XML or JSON format, the device omits the root `configuration` object in the following cases:

- The comparison returns no differences

- The comparison returns differences for only non-native configuration data, for example, configuration data associated with an OpenConfig data model.

`database` Specify the configuration database from which to display data.

- **Default:** `candidate`
- **Values:**
  - `candidate`—The candidate configuration
  - `committed`—The active configuration (the one most recently committed)

If you include both the `database` and the `database-path` attributes, the `database` attribute takes precedence.

`database-path`

Within a commit script, this attribute specifies the path to the session's pre-inheritance candidate configuration. For normal configuration sessions, the commit script retrieves the normal, pre-inheritance candidate configuration. For private configuration sessions, the commit script retrieves the private, pre-inheritance candidate configuration.

- **Values:** `$junos-context/commit-context/database-path`

If you include both the `database` and the `database-path` attributes, the `database` attribute takes precedence.

`format`

Specify the format in which the NETCONF or Junos XML protocol server returns the configuration data.

- **Default:** `xml`
- **Values:**
  - `json`—Configuration data format is JSON.

    NOTE: Integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks.

  - `set`—Configuration data format is Junos OS configuration mode commands.
  - `text`—Configuration data format is ASCII text, which uses the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements.
  - `xml`—Configuration data format is Junos XML.

NOTE: Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 22.3R1, NETCONF sessions additionally support the `json-minified` and `xml-`

`minified` formats, which return the respective format with unnecessary spaces, tabs, and newlines removed.

`groups` Request that the `junos:group="`*`group-name`*`"` attribute appear in the opening tag for each configuration element that is inherited from a configuration group. The *group-name* variable specifies the name of the configuration group from which that element was inherited.

- **Values:** `groups`

When you specify the `groups` attribute, you must also specify the `inherit` attribute.

`inherit` Specify how the NETCONF or Junos XML protocol server displays statements that are defined in configuration groups and interface ranges. If you omit the `inherit` attribute, the output uses the `<groups>`, `<apply-groups>`, and `<apply-groups-except>` tags to represent user-defined configuration groups and uses the `<interface-range>` tag to represent user-defined interface ranges. The output does not include tag elements for statements defined in the junos-defaults group.

- **Values:**
  - `defaults`—The output does not include the `<groups>`, `<apply-groups>`, and `<apply-groups-except>` tags, but instead displays elements that are inherited from user-defined groups and from the junos-defaults group as children of the inheriting tag elements.
  - `inherit`—The output does not include the `<groups>`, `<apply-groups>`, `<apply-groups-except>`, and `<interface-range>` tags, but instead displays elements that are inherited from user-defined groups and ranges as children of the inheriting tag elements. The output does not include tag elements for statements defined in the junos-defaults group.

`interface-ranges` Request that the `junos:interface-ranges="`*`source-interface-range`*`"` attribute appear in the opening tag for each configuration element that is inherited from an interface range. The *source-interface-range* variable specifies the name of the interface range.

- **Values:** `interface-ranges`

When you specify the `interface-ranges` attribute, you must also specify the `inherit` attribute.

`junos:key | key` Request that the `junos:key="key"` attribute appear in the opening tag of each element that serves as an identifier for a configuration object.

- **Values:** `key`

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

`interface-ranges` attribute added in Junos OS Release 10.3R1.

`commit-scripts` attribute values `apply` and `apply-no-transients` added in Junos OS Release 12.1

`database-path` attribute added in Junos OS Release 12.2.

`format` attribute value `json` added in Junos OS Release 14.2.

`format` attribute value `set` added in Junos OS Release 15.1.

Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks.

`compare` attribute value `configuration-revision` added in Junos OS Release 20.4R1 and Junos OS Evolved Release 20.4R1.

`format` attribute values `json-minified` and `xml-minified` added for NETCONF sessions only in Junos OS Release 21.1R1 and Junos OS Evolved Release 22.3R1.

RELATED DOCUMENTATION

Request Configuration Data Using the Junos XML Protocol

junos:changed

junos:group

junos:interface-range

junos:key

<load-configuration>

IN THIS SECTION

Usage

```
<rpc>
    <load-configuration configuration-revision="revision-id"/>

    <load-configuration rescue="rescue"/>

    <load-configuration rollback="index"/>

    <load-configuration url="url"
           [action="(merge | override | replace | update)"]
            [format="(text | xml)"] />

    <load-configuration url="url" [action="(merge | override | update)"]
            format="json" />

    <load-configuration url="url" action="set" format="text"/>

    <load-configuration [action="(merge | override | replace | update)"]
            [format="xml"]  >
        <configuration>
            <!-- tag elements for configuration elements to load -->
        </configuration>
    </load-configuration>

    <load-configuration [action="(merge | override | replace | update)"]
            format="text" >
        <configuration-text>
```

```
            <!-- formatted ASCII configuration statements to load -->
        </configuration-text>
    </load-configuration>

    <load-configuration [action="(merge | override | update)"] format="json">
        <configuration-json>
            <!-- JSON configuration data to load -->
        </configuration-json>
    </load-configuration>

    <load-configuration action="set" format="text" >
        <configuration-set>
            <!-- configuration mode commands to load -->
        </configuration-set>
    </load-configuration>
</rpc>
```

Description

Request that the NETCONF or Junos XML protocol server load configuration data into the candidate configuration or open configuration database.

If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<load-configuration>` operation, the server loads the configuration data into the open configuration database. Otherwise, the server loads the configuration data into the candidate configuration.

Table 15 on page 247 describes the common load configuration operations.

Table 15: Load Configuration Data

`<load-configuration>` Operation	Description
`<load-configuration configuration-revision="`*`revision-id`*`"/>`	Load a previously committed configuration by referencing its configuration revision ID. The specified configuration completely replaces the candidate configuration.
`<load-configuration rescue="rescue"/>`	Load the rescue configuration. The rescue configuration completely replaces the candidate configuration.

Table 15: Load Configuration Data *(Continued)*

`<load-configuration>` Operation	Description
`<load-configuration rollback="`*`index`*`"/>`	Load a previously committed configuration by referencing its numerical rollback index. The specified configuration completely replaces the candidate configuration.

`<load-configuration url="`*`url`*`" format="(text | xml | json)/>`
 	
`<load-configuration url="`*`url`*`" action="set" format="text"/>`
	Load configuration data from the file specified in the `url` attribute. Specify the full path of the file that contains the configuration data to load and the format of the data in the file. For example:
	
	`<load-configuration url="/tmp/add.conf" format="text"/>`

`<load-configuration format="xml">`
`    <configuration>...</configuration>`
`</load-configuration>`
 	
`<load-configuration format="text">`
`    <configuration-text>...</configuration-text>`
`</load-configuration>`
 	
`<load-configuration format="json">`
`    <configuration-json>...</configuration-json>`
`</load-configuration>`
 	
`<load-configuration action="set" format="text">`
`    <configuration-set>...</configuration-set>`
`</load-configuration>`
	Load the configuration as a data stream. Enclose the configuration data in the appropriate set of tags for the format.

Attributes

`action`

Specify how to load the configuration data, particularly when the target configuration database and the loaded configuration contain conflicting statements.

The ephemeral configuration database supports all of the `action` attribute values. The `update` value is supported in Junos OS Release 21.1R1 and later.

- **Default:** `merge`

- **Values:**

  - `merge`—Combine the data in the loaded configuration with the data in the target configuration. If statements in the loaded configuration conflict with statements in the target configuration, the loaded statements replace those statements in the target configuration.

  - `override`—Discard the entire candidate configuration and replace it with the loaded configuration. When you commit the configuration, all system processes parse the new configuration.

  - `replace`—Substitute each hierarchy level or configuration object defined in the loaded configuration for the corresponding level or object in the candidate configuration.

    If the configuration data format is ASCII text, place the `replace:` statement on the line directly preceding the statements that represent the hierarchy level or object to replace. If the configuration data is Junos XML elements, include the `replace="replace"` attribute in the opening tags of the elements that represent the hierarchy levels or objects to replace.

  - `set`—Load configuration data formatted as Junos OS configuration mode commands. This option executes the configuration instructions line by line. You can store the instructions in a file named by the `url` attribute, or you can enclose the instructions in a `<configuration-set>` element to provide a data stream. The instructions can contain any configuration mode command, such as `set`, `delete`, `edit`, or `deactivate`. When using the `set` action, the default and only acceptable value for the `format` attribute is "text".

  - `update`—Compare a complete loaded configuration against the candidate configuration. For each hierarchy level or configuration object that is different in the two configurations, the version in the loaded configuration replaces the version in the candidate configuration. When you commit the configuration, only system processes that are affected by the changed configuration elements parse the new configuration.

`configuration-revision` Load a previously committed configuration by referencing its configuration revision ID. The specified configuration completely replaces the candidate configuration.

`format` Specify the format used for the configuration data.

- **Default:**

- `xml`, for all `action` values except `set`
- `text`, when `action="set"`

- Values:
  - `json`–Indicate that the configuration data format is JSON.
  - `text`—Indicate that the configuration data format is ASCII text or configuration mode commands.

    ASCII text format uses the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements. Junos devices use this format for configuration files stored on the device and for the output of the CLI `show configuration` command.

    The `set` command format consists of Junos OS configuration mode commands. You can view this format using the `show configuration | display set` CLI command. To load configuration mode commands, you must set the `action` attribute to "set".
  - `xml`—Indicate that the configuration data is Junos XML elements.

`rescue` Replace the candidate configuration with the rescue configuration.

- Values: `rescue`

> **NOTE**: You can also use the `<rollback-config>` RPC to load a previously committed configuration. The `<rollback-config>` RPC is useful for applications that do not support executing RPCs that include XML attributes.

`rollback` Load a previously committed configuration by referencing its numerical rollback index. Valid values are 0 (for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49).

> **NOTE**: You can also use the `<rollback-config>` RPC to load a previously committed configuration. The `<rollback-config>` RPC is useful for applications that do not support executing RPCs that include XML attributes.

`url` Specify the full pathname of the file that contains the configuration data to load. The value can be a local file path, an FTP location, or an HTTP URL.

- **Syntax:**
  - Local filename:
    - ***/path/ filename***—File on a mounted file system, either on the local flash drive or on hard disk.
    - **a:*filename*** or **a:*path/ filename***—File on the local drive. The default path is **/** (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.
  - **ftp://*username:password@hostname/ path/ filename***
  - **http://*username:password@hostname/ path/ filename***

In each case, the default value for the ***path*** variable is the home directory for the username. To specify an absolute path, the application starts the path with the characters **%2F**; for example, **ftp://*username:password@hostname/*%2F*path/ filename***.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

`action` attribute value `set` added in Junos OS Release 11.4.

`format` attribute value `json` added in Junos OS Release 16.1.

`configuration-revision` attribute added in Junos OS Release 20.4R1 and Junos OS Evolved Release 20.4R1.

RELATED DOCUMENTATION

Request Configuration Changes Using the Junos XML Protocol

<load-configuration-results>

replace

<load-configuration-results>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <load-configuration-results>
        <load-success/>
        <load-error-count>errors</load-error-count>
    </load-configuration-results>
</rpc-reply>
```

Description

Tag element returned by the NETCONF or Junos XML protocol server in response to a `<load-configuration>` request by a client application.

In a Junos XML protocol session, the `<load-configuration-results>` element encloses either a `<load-success/>` tag or a `<load-error-count>` tag, which indicates the success or failure of the load configuration operation. In a NETCONF session, the `<load-configuration-results>` element encloses either an `<ok/>` tag or a `<load-error-count>` tag to indicate the success or failure of the load configuration operation.

Contents

`<load-error-count>`	Specifies the number of errors that occurred when the server attempted to load new data into the candidate configuration or open configuration database. The target configuration must be restored to a valid state before it is committed.
`<load-success/>`	Indicates that the server successfully loaded new data into the candidate configuration or open configuration database.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<load-configuration>

<lock-configuration/>

IN THIS SECTION

Usage

```
<rpc>
    <lock-configuration/>
</rpc>
```

Description

Request that the NETCONF or Junos XML protocol server open and lock the candidate configuration. This operation enables the client application to read and change the candidate configuration while preventing other users or applications from changing it. The application must emit the `<unlock-configuration/>` tag to unlock the configuration.

If the Junos XML protocol session ends or the application emits the `<unlock-configuration/>` tag before the candidate configuration is committed, all changes made to the candidate are discarded.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Lock, Unlock, or Create a Private Copy of the Candidate Configuration Using the Junos XML Protocol

<unlock-configuration/>

<open-configuration>

IN THIS SECTION

Usage

```
<rpc>
    <open-configuration>
        <private/>
    </open-configuration>

    <open-configuration>
        <ephemeral/>
    </open-configuration>

    <open-configuration>
        <ephemeral-instance>instance-name</ephemeral-instance>
    </open-configuration>
</rpc>
```

Description

Create a private copy of the candidate configuration database or open the default instance or a user-defined instance of the ephemeral configuration database.

NOTE: Before opening a user-defined instance of the ephemeral configuration database, you must first enable the instance by configuring the `instance` *`instance-name`* statement at the `[edit system configuration-database ephemeral]` hierarchy level on the device.

A client application can perform the same operations on the private copy or ephemeral instance as on the regular candidate configuration, including load and commit operations. There are, however, restrictions on these operations. For details, see *<load-configuration>* and *<commit-configuration>*.

To close a private copy or ephemeral instance and discard all uncommitted changes, execute the `<close-configuration/>` operation. Changes to the private copy or ephemeral instance are also lost if the NETCONF or Junos XML protocol session ends for any reason before the changes are committed. It is not possible to save the changes other than by performing a commit operation, for example, by emitting the `<commit-configuration/>` tag.

NOTE: The Junos XML protocol `<open-configuration>` operation does not emit an "`uncommitted changes will be discarded on exit`" warning message when opening a private copy of the candidate configuration. However, the device still discards the uncommitted changes upon closing the private copy.

Contents

`<private/>`	Open a private copy of the candidate configuration database.
`<ephemeral/>`	Open the default instance of the ephemeral configuration database.
`<ephemeral-instance>`	Open the specified instance of the ephemeral configuration database. This instance must already be configured at the `[edit system configuration-database ephemeral]` hierarchy level on the device.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

`<ephemeral>` and `<ephemeral-instance>` elements added in Junos OS Release 16.2R2.

RELATED DOCUMENTATION

Lock, Unlock, or Create a Private Copy of the Candidate Configuration Using the Junos XML Protocol

<reason>

IN THIS SECTION

Usage

```
<xnm:error | xnm:warning>
    <reason>
        <daemon>process</daemon>
        <process-not-configured/>
        <process-disabled/>
        <process-not-running/>
    </reason>
</xnm:error | xnm:warning>
```

Description

Child element included in an `<xnm:error>` or `<xnm:warning>` element in a NETCONF protocol server response to explain why a process could not service a request.

Contents

`<daemon>`	Identifies the process.
`<process-disabled>`	Indicates that the process has been explicitly disabled by an administrator.
`<process-not-configured>`	Indicates that the process has been disabled because it is not configured.
`<process-not-running>`	Indicates that the process is not running.

Release Information

This is a Junos XML management protocol response tag. It is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<xnm:error> | 261

<xnm:warning> | 264

<request-end-session/>

IN THIS SECTION

Usage

```
<rpc>
    <request-end-session/>
</rpc>
```

Description

Request that the NETCONF or Junos XML protocol server end the current session.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<end-session/>

<routing-engine>

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <commit-results>

    <!-- when the candidate configuration or private copy is committed -->
        <routing-engine>
            <name>reX</name>
            <commit-success/>
            <commit-revision-information>
                <old-db-revision>old-revision-id</old-db-revision>
                <new-db-revision>new-revision-id</new-db-revision>
            </commit-revision-information>
        </routing-engine>

    <!-- when the candidate configuration or private copy is syntactically valid -->
        <routing-engine>
            <name>reX</name>
            <commit-check-success/>
        </routing-engine>

    <!-- when an instance of the ephemeral database is committed -->
        <routing-engine>
            <name>reX</name>
            <commit-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

Description

Child element included in a Junos XML protocol server `<commit-results>` response element to return information about a requested commit operation on a particular Routing Engine.

Contents

`<commit-check-success>`	Indicates that the configuration is syntactically correct.
`<commit-success>`	Indicates that the Junos XML protocol server successfully committed the configuration.

`<name>`	Name of the Routing Engine on which the commit operation was performed. Possible values are re0 and re1.

The `<commit-revision-information>` tag element is described separately.

Release Information

This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<commit-results>

<commit-revision-information>

<unlock-configuration/>

IN THIS SECTION

Usage

```
<rpc>
    <unlock-configuration/>
</rpc>
```

Description

Request that the NETCONF or Junos XML protocol server unlock and close the candidate configuration. Until the application emits this tag, other users or applications can read the configuration but cannot change it.

Release Information

This operation is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

Lock, Unlock, or Create a Private Copy of the Candidate Configuration Using the Junos XML Protocol

<lock-configuration/>

<xnm:error>

IN THIS SECTION

Usage

```
<xnm:error xmlns="namespace-URL" xmlns:xnm="namespace-URL">
    <parse/>
    <source-daemon>module-name </source-daemon>
    <filename>filename</filename>
    <line-number>line-number </line-number>
    <column>column-number</column>
    <token>input-token-id </token>
    <edit-path>edit-path</edit-path>
    <statement>statement-name </statement>
    <message>error-string</message>
    <re-name>re-name-string</re-name>
    <database-status-information>...</database-status-information>
    <reason>...</reason>
</xnm:error>
```

Description

Indicates that the NETCONF server has experienced an error while processing the client application's request. If the server has already emitted the response tag element for the current request, the information enclosed in the response tag element might be incomplete. The client application must include code that discards or retains the information, as appropriate. The child tag elements detail the nature of the error. The NETCONF server does not necessarily emit all child tag elements; it emits tag elements that are relevant to the current request.

Attributes

xmlns	Defines the XML namespace for the contents of the tag element. The value is a URL of the form `http://xml.juniper.net/xnm/`*`version`*`/xnm`, where *version* is a string such as 1.1.
xmlns:xnm	Defines the XML namespace for child tag elements that include the `xnm:` prefix. The value is a URL of the form `http://xml.juniper.net/xnm/`*`version`*`/xnm`, where *version* is a string such as 1.1.

Contents

`<column>`	(Occurs only during loading of a configuration file) Identifies the element that caused the error by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are defined in the `<line-number>` and `<filename>` elements.
`<edit-path>`	(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the error occurred, in the form of the CLI configuration mode banner.
`<filename>`	(Occurs only during loading of a configuration file) Filename of the configuration file that was being loaded.

`<line-number>`	(Occurs only during loading of a configuration file) Specifies the line number where the error occurred in the configuration file that was being loaded, which is defined in the `<filename>` element.
`<message>`	Describes the error in a natural-language text string.
`<parse/>`	Indicates that there was a syntactic error in the request submitted by the client application.
`<re-name>`	Identifies the Routing Engine on which the error occurred.
`<source-daemon>`	Identifies the Junos OS module that was processing the request in which the error occurred.
`<statement>`	(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The `<edit-path>` element specifies the statement's parent hierarchy level.
`<token>`	Identifies the element in the request that caused the error.

The other tag elements are explained separately.

Release Information

This is a Junos XML management protocol response tag. It is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<database-status-information> | **236**

<reason> | **256**

<xnm:warning> | **264**

<xnm:warning>

IN THIS SECTION

Usage

```
<xnm:warning xmlns="namespace-URL" xmlns:xnm="namespace-URL">
    <source-daemon>module-name </source-daemon>
    <filename>filename</filename>
    <line-number>line-number </line-number>
    <column>column-number</column>
    <token>input-token-id </token>
    <edit-path>edit-path</edit-path>
    <statement>statement-name </statement>
    <message>error-string</message>
    <reason>...</reason>
</xnm:warning>
```

Description

Indicates that the server has encountered a problem while processing the client application’s request. The child tag elements detail the nature of the warning.

Attributes

`xmlns`—Defines the XML namespace for the contents of the tag element. The value is a URL of the form `http://xml.juniper.net/xnm/`*`version`*`/xnm`, where *`version`* is a string such as 1.1.

`xmlns:xnm`—Defines the XML namespace for child tag elements that have the `xnm:` prefix in their names. The value is a URL of the form `http://xml.juniper.net/xnm/`*`version`*`/xnm`, where *`version`* is a string such as 1.1.

Contents

`<column>`	(Occurs only during loading of a configuration file) Identifies the element that caused the problem by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are defined in the `<line-number>` and `<filename>` tag elements.
`<edit-path>`	(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the problem occurred, in the form of the CLI configuration mode banner.
`<filename>`	(Occurs only during loading of a configuration file) Filename of the configuration file that was being loaded.
`<line-number>`	(Occurs only during loading of a configuration file) Specifies the line number where the problem occurred in the configuration file that was being loaded, which is defined in the `<filename>` element.
`<message>`	Describes the warning in a natural-language text string.
`<source-daemon>`	Identifies the Junos OS module that was processing the request in which the warning occurred.
`<statement>`	(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The `<edit-path>` element specifies the statement's parent hierarchy level.
`<token>`	Identifies the element in the request that caused the warning.

The other tag element is explained separately.

Release Information

This is a Junos XML management protocol response tag. It is supported as a Juniper Networks proprietary extension in NETCONF sessions on Junos devices that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.

RELATED DOCUMENTATION

<reason> | 256

<xnm:error> | 261

CHAPTER 9

Junos XML Protocol Element Attributes Supported in NETCONF Sessions

IN THIS CHAPTER

junos:changed-localtime

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:changed-seconds="seconds" \
        junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ">
        <!-- Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>
```

Description

(Displayed when the candidate configuration is requested) Specifies the time when the configuration was last changed as the date and time in the device's local time zone.

Usage Guidelines

See "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | 216

junos:changed-seconds | 267

xmlns | 273

junos:changed-seconds

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:changed-seconds="seconds" \
         junos:changed-localtime="YYY-MM-DD hh:mm:ss TZ">
        <!-- Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>
```

Description

(Displayed when the candidate configuration is requested) Specifies the time when the configuration was last changed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines

See "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | **216**

junos:changed-localtime | **266**

xmlns | **273**

junos:commit-localtime

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
     <configuration xmlns="URL" junos:commit-seconds="seconds" \
        junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
        junos:commit-user="username">
        <!-- Junos XML tag elements for the requested configuration data -->
```

```
    </configuration>
</rpc-reply>
```

Description

(Displayed when the active configuration is requested) Specifies the time when the configuration was committed as the date and time in the device’s local time zone.

Usage Guidelines

See "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | **216**

junos:commit-user | **270**

junos:commit-seconds | **269**

xmlns | **273**

junos:commit-seconds

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:commit-seconds="seconds" \
        junos:commit-localtime="YYY-MM-DD hh:mm:ss TZ" \
        junos:commit-user="username">
        <!--Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>
```

Description

(Displayed when the active configuration is requested) Specifies the time when the configuration was committed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines

See "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | **216**

junos:commit-user | **270**

junos:commit-localtime | **268**

xmlns | **273**

junos:commit-user

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:commit-seconds="seconds" \
        junos:commit-localtime="YYY-MM-DD hh:mm:ss TZ" \
        junos:commit-user="username">
        <!-- Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>
```

Description

(Displayed when the active configuration is requested) Specifies the Junos OS username of the user who requested the commit operation.

Usage Guidelines

See "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | 216

junos:commit-localtime | 268

junos:commit-seconds | 269

xmlns | 273

replace-pattern

IN THIS SECTION

Usage

```
<rpc>
   <load-configuration>

      <!-- replace a pattern  globally  -->
      <configuration replace-pattern="pattern1" with="pattern2" [upto="n"]>
      </configuration>

      <!-- replace a pattern at a specific hierarchy level  -->
      <configuration>
         <!-- opening tag for each parent element -->
            <level-or-object replace-pattern="pattern1" with="pattern2"
                  [upto="n"]/>
         <!-- closing tag for each parent element -->
      </configuration>

      <!-- replace a pattern for an object that has an identifier -->
      <configuration>
         <!-- opening tag for each parent  element -->
            <container-tag replace-pattern="pattern1" with="pattern2"
                  [upto="n"]>
                <name>identifier</name>
            </container-tag>
         <!-- closing tag for each parent element -->
      </configuration>

   </load-configuration>
</rpc>
```

Description

Replace a variable or identifier in the candidate configuration or open configuration database. Junos OS replaces the pattern specified by the `replace-pattern` attribute with the replacement pattern defined by the `with` attribute. The optional `upto` attribute limits the number of objects replaced. The placement of the attributes in the configuration data determines the scope of the replacement.

Attributes

`replace-pattern="`*`pattern1`*`"`	Text string or regular expression that defines the identifiers or values you want to match.
`with="`*`pattern2`*`"`	Text string or regular expression that replaces the identifiers and values located with *pattern1*.
`upto="`*`n`*`"`	Number of objects replaced. The value of *n* controls the total number of objects that the device replaces in the configuration (not the total number of times the pattern occurs). The device replaces objects at the same hierarchy level (siblings) first. The device considers multiple occurrences of a pattern within a given object as a single replacement. If you omit the `upto` attribute or if you set the attribute equal to zero, the device replaces all identifiers and values that match the pattern. • **Range:** 1 through 4294967295 • **Default:** 0

Release Information

Attribute introduced in Junos OS Release 15.1R1.

RELATED DOCUMENTATION

Replace Patterns in Configuration Data Using the NETCONF or Junos XML Protocol

Modifying the Configuration for a Device

Modifying the Configuration for a Device

replace

xmlns

IN THIS SECTION

Usage

```
<rpc-reply xmlns:junos="URL">
    <operational-response xmlns="URL-for-DTD">
        <!-- Junos XML tag elements for the requested operational data -->
    </operational-response>
</rpc-reply>

<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:(changed | commit)-seconds="seconds" \
                junos:(changed | commit)-localtime="YYY-MM-DD hh:mm:ss TZ" \
                [junos:commit-user="username"]>
        <!-- Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>
```

Description

For operational responses, defines the XML namespace for the enclosed tag elements that do not have a prefix (such as `junos:`) in their names. The namespace indicates which Junos XML document type definition (DTD) defines the set of tag elements in the response.

For configuration data responses, define the XML namespace for the enclosed tag elements.

Usage Guidelines

See "Request Operational Information Using NETCONF" on page 386 and "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

RELATED DOCUMENTATION

<rpc-reply> | 216

junos:changed-localtime | 266

junos:changed-seconds | 267

junos:commit-user | **270**

junos:commit-localtime | **268**

junos:commit-seconds | **269**

3

PART

Manage Configurations Using NETCONF

CHAPTER 10

Change the Configuration Using NETCONF

IN THIS CHAPTER

Edit the Configuration Using NETCONF

In a NETCONF session with a Junos device, you can use NETCONF operations to modify the device configuration. The NETCONF operations `<copy-config>`, `<edit-config>`, and `<discard-changes>` offer functionality that is analogous to configuration mode commands in the Junos OS CLI. The `<copy-config>` and `<edit-config>` operations support loading configuration data formatted as Junos XML elements or CLI configuration statements.

To change the configuration, a client application emits the `<copy-config>`, the `<edit-config>`, or the `<discard-changes>` tag element and the corresponding tag subelements within the `<rpc>` tag element.

The following examples show the various available tag elements:

```
<rpc>
    <copy-config>
        <target><candidate/></target>
        <error-operation> (ignore-error | stop-on-error) </error-operation>
        <source>
            (<running/> | <url>location</url>)
        </source>
    </copy-config>
</rpc>
]]>]]>
```

```
<rpc>
     <edit-config>
        <target><candidate/></target>
        <default-operation>operation</default-operation>
        <error-operation>error</error-operation>
        <(config | config-text | url)>
            <!-- configuration change file or data -->
        </(config | config-text | url)>
    </edit-config>
</rpc>
]]>]]>
```

```
<rpc>
    <discard-changes/>
</rpc>
]]>]]>
```

The only acceptable value for the `<target>` element is `<candidate/>`, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing a `<copy-config>` or `<edit-config>` operation, the device performs the operation on the open configuration database. Otherwise, the device performs the operation on the candidate configuration.

The three tags—`<copy-config>`, `<edit-config>`, and `<discard-changes>`—correspond to the three basic configuration tasks available to you, which are described here:

- Overwrite the target configuration with a new configuration—Use the `<copy-config>` operation to replace the target configuration with a new configuration.
- Edit configuration elements—Use the `<edit-config>` operation to add, change, or delete specific configuration elements within the target configuration. To specify how the device should handle configuration changes, see "Set the Edit Configuration Mode in a NETCONF Session" on page 287.
- Roll back changes to the current configuration—Use the `<discard-changes>` operation to roll back the candidate configuration to match the contents of the current running (active) configuration. This operation is analogous to the `rollback 0` configuration mode command in the CLI.

  NOTE: The `<discard-changes/>` tag element cannot be used to discard uncommitted changes that have been loaded into the ephemeral configuration database.

RELATED DOCUMENTATION

Upload and Format Configuration Data in a NETCONF Session

IN THIS SECTION

A NETCONF client application can specify the delivery mechanism and the format of the configuration data when modifying the Junos device configuration. Client applications can use a text file or streaming data to upload configuration data in one of the accepted formats to the candidate configuration or open configuration database.

A client can choose to stream configuration changes within the session or reference data files that include the desired configuration changes. Each method has advantages and disadvantages. Streaming data allows you to send your configuration change data in line, using your NETCONF connection. This is useful when the device is behind a firewall and you cannot establish another connection to upload a data file. With text files you can keep the edit configuration commands simple; there is no need to include the possibly complex configuration data stream.

The `<copy-config>` and `<edit-config>` operations accept one of two formats for the Junos configuration data: Junos XML or CLI configuration statements. The choice between one data format over the other is personal preference.

NOTE: When managing Junos devices, a NETCONF client can use the Junos XML protocol `<load-configuration>` operation in a NETCONF session to upload configuration data formatted using JSON or configuration mode `set` commands, in addition to Junos XML or CLI configuration statement formats.

The delivery mechanism and the format are discussed in detail in the following sections.

Referencing Configuration Data Files

A client application can use the `<copy-config>` or `<edit-config>` operation to upload configuration data stored in a file. The `<copy-config>` operation replaces the entire configuration. The `<edit-config>` operation loads the data as indicated by the edit configuration mode specified in the `<default-operation>` element or the individual configuration data elements.

Before loading the file, the client application or an administrator saves Junos XML tag elements or CLI configuration statements as the contents of the file. The file includes the tag elements or configuration statements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to each element to change.

To specify the file, the client includes the `<url>` element as shown in the examples.

```
<rpc>
    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
```

```
            <url>
               <!-- location and name of file containing configuration data -->
            </url>
        </source>
    </copy-config>
</rpc>
]]>]]>
```

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <url>
           <!-- location and name of file containing configuration data -->
        </url>
    </edit-config>
</rpc>
]]>]]>
```

The data within these files can be formatted as either Junos XML elements or CLI configuration statements. When the configuration data is formatted as CLI configuration statements, include the `format="text"` attribute in the `<url>` tag.

```
<url format="text">
    <!-- location and name of file containing configuration data -->
</url>
```

The configuration file can be placed locally or as a network resource.

- When placed locally, the configuration file path can be relative or absolute.
    - Relative file path—The file location is relative to the user's home directory. For example:

      ```
      <url>config-replace.xml</url>
      ```

- Absolute file path—The file location is based on the directory structure of the device, for example ***<drive>:filename*** or ***<drive>/:path/ filename*** . If you are using removable media, the drive can be in the MS-DOS or UNIX (UFS) format. Junos OS also supports the `file://` URI. For example:

  ```
  <url>/var/tmp/config-replace.xml</url>
  ```

  ```
  <url>file:///var/tmp/config-replace.xml</url>
  ```

- When located on the network, the configuration file can be accessed using FTP or HTTP:
  - FTP example:

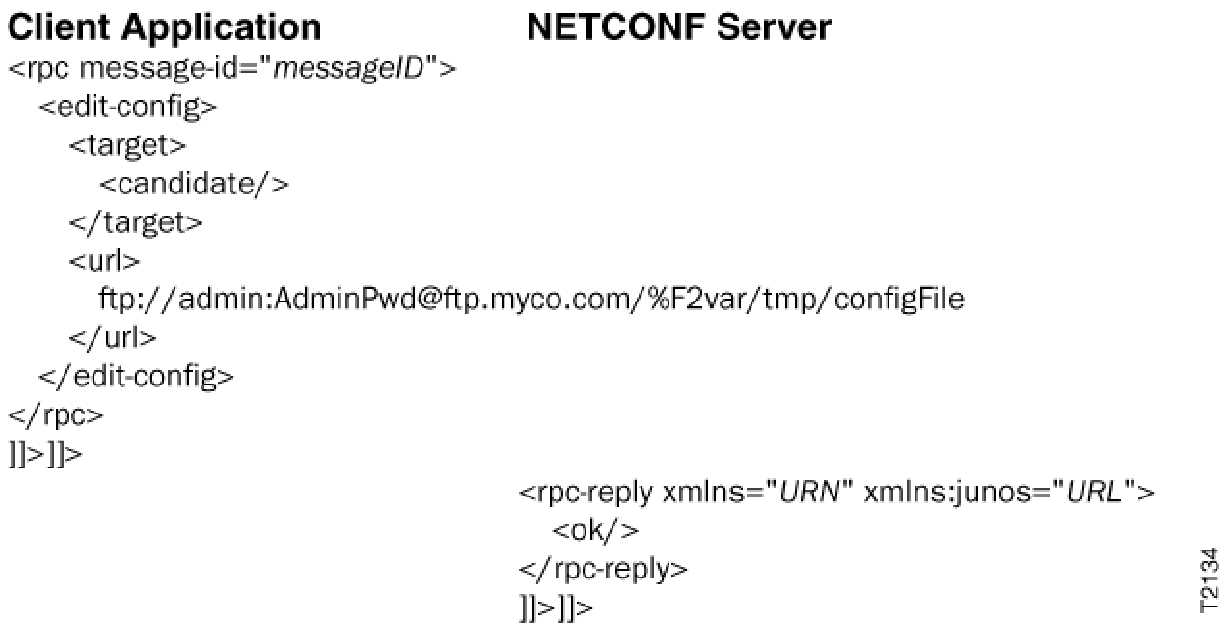
    ```
    ftp://username:password@hostname/path/filename
    ```

    NOTE: By default, the FTP ***path*** variable is relative to the user's home directory. To specify an absolute path, start the path with the characters %2F; for example: **ftp://*username*:*password*@*hostname*/%2F*path*/ *filename*** .

  - HTTP example:

    ```
    http://username:password@hostname/path/filename
    ```

The following example shows how to incorporate configuration data stored in the file **/var/tmp/configFile** on the FTP server called **ftp.myco.com**:



```
<rpc message-id="messageID">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <url>
      ftp://admin:AdminPwd@ftp.myco.com/%F2var/tmp/configFile
    </url>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

Streaming Configuration Data

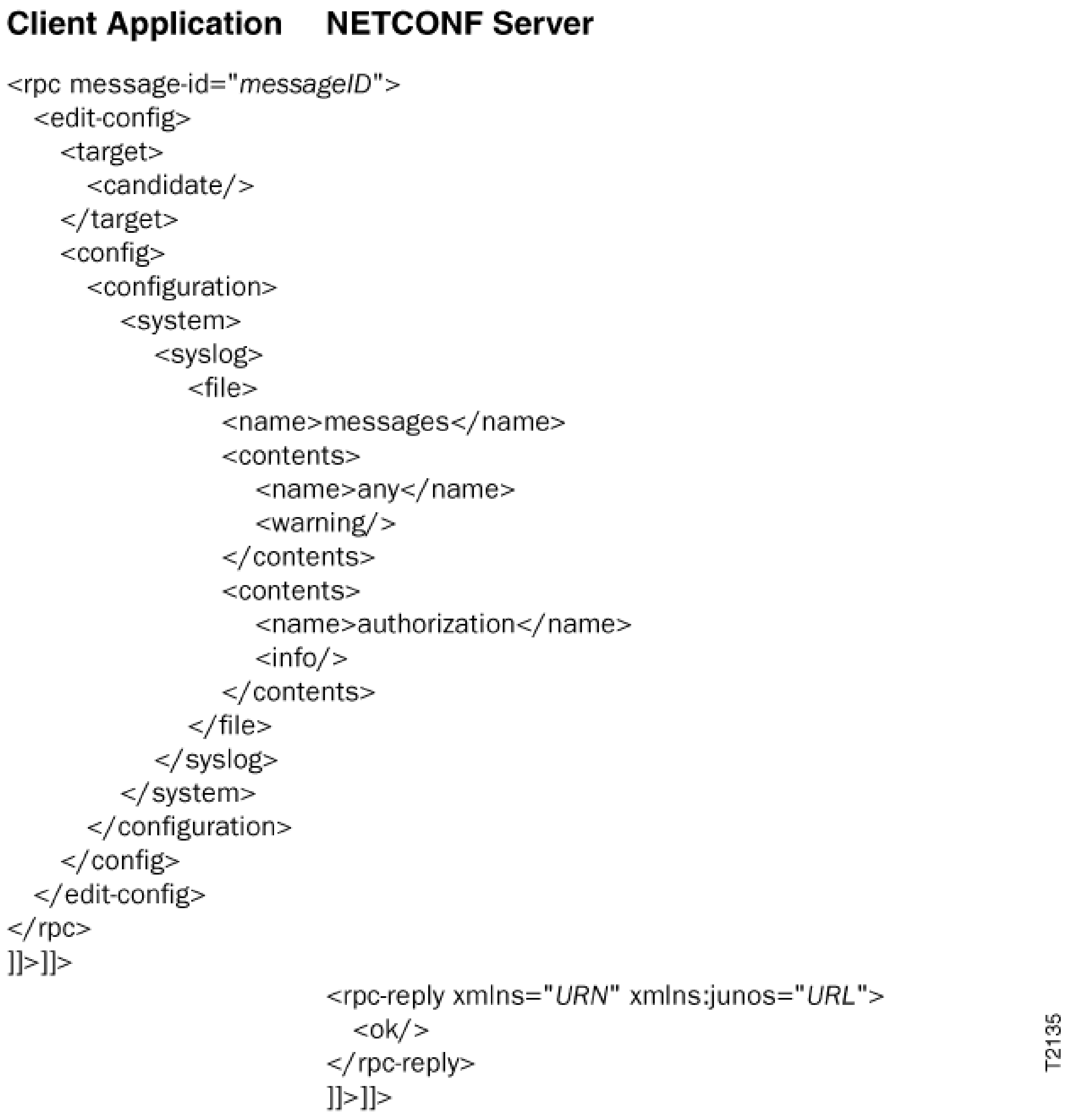
To provide configuration data as a data stream, a client application emits the `<config>` or `<config-text>` tag elements within the `<rpc>` and `<edit-config>` tag elements. To specify the configuration elements to change, the application emits Junos XML or CLI configuration statements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` or `<configuration-text>` tag element) down to each element to change.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <config>
            <configuration>
                <!-- configuration changes -->
            </configuration>
        </config>
    </edit-config>
</rpc>
]]>]]>
```

```
<rpc>
    <edit-config>
```

```
        <target>
            <candidate/>
        </target>
        <config-text>
            <configuration-text>
                <!-- configuration changes -->
            </configuration-text>
        </config-text>
    </edit-config>
</rpc>
]]>]]>
```

The following example shows how to provide Junos XML configuration data in a data stream to configure the **messages** system log file:



```
<rpc message-id="messageID">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <syslog>
            <file>
              <name>messages</name>
              <contents>
                <name>any</name>
                <warning/>
              </contents>
              <contents>
                <name>authorization</name>
                <info/>
              </contents>
            </file>
          </syslog>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

Formatting Data: Junos XML versus CLI Configuration Statements

The NETCONF `<copy-config>` and `<edit-config>` operations accept one of two formats for Junos OS configuration data: Junos XML or CLI configuration statements. The choice between one data format over the other is personal preference.

NOTE: When managing Junos devices, a NETCONF client can use the Junos XML protocol `<load-configuration>` operation in a NETCONF session to upload configuration data formatted using JSON or configuration mode `set` commands, in addition to Junos XML or CLI configuration statement formats.

When you load configuration data from a file, you include the `<url>` element with the file location. The default format for the configuration data is Junos XML.

```
<url>dataFile</url>
```

To specify that the data file contains CLI configuration statements, include the `format="text"` attribute in the `<url>` tag.

```
<url format="text">dataFile</url>
```

When streaming data, you specify the data format by including one of two tags: `<config>` for Junos XML elements or `<config-text>` for CLI configuration statements.

In the following example, the `<configuration>` element encloses Junos XML-formatted configuration data:

```
<config>
  <configuration>
    <system>
      <services>
        <ssh>
           <protocol-version>v2</protocol-version>
        </ssh>
      </services>
    </system>
  </configuration>
</config>
```

In the following example, the `<configuration-text>` element encloses the same data formatted as CLI configuration statements:

```
<config-text>
  <configuration-text>
    system {
      services {
        ssh {
           protocol-version v2;
        }
      }
    }
  </configuration-text>
</config-text>
```

RELATED DOCUMENTATION

Edit the Configuration Using NETCONF | 277

Map Configuration Statements to Junos XML Tag Elements | 25

<copy-config> | 186

<edit-config> | 191

Set the Edit Configuration Mode in a NETCONF Session

IN THIS SECTION

When sending configuration data to the NETCONF server, you can specify how the device should handle the configuration changes. This is known as the edit configuration mode. You can set the edit configuration mode globally for the entire session. You can also set the edit mode for only specific elements within the session.

Devices running Junos OS have the following edit configuration modes:

- `merge`—The device merges new configuration data into the existing configuration data. This is the default.
- `replace`—The device replaces existing configuration data with the new configuration data.
- `none`—The device does not change the existing configuration unless the new configuration element includes an operation attribute.

To set the edit configuration mode globally for the session, include the `<default-operation>` element with the desired mode as a child element of `<edit-config>`.

```
<rpc>
  <edit-config>
    <default-operation>mode</default-operation>
  <edit-config>
</rpc>
```

To specify the edit configuration mode for an individual element, include the `operation` attribute and desired mode in that element's tag.

```
<rpc>
  <edit-config>
    <config>
      <configuration>
        <protocols>
          <rip>
            <message-size operation="replace">255</message-size>
          </rip>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

You can also set a global edit configuration mode for an entire set of configuration changes and specify a different mode for individual elements that you want handled in a different manner. For example:

```
<rpc>
  <edit-config>
    <default-operation>merge</default-operation>
    <config>
      <configuration>
        <protocols>
          <rip>
            <message-size operation="replace">255</message-size>
          </rip>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

The edit configuration modes are discussed in more detail in the following sections:

Specifying the merge Data Mode

By default, the NETCONF server *merges* new configuration data into the candidate configuration or open configuration database. Thus, if you do not specify an edit configuration mode, the device merges the new configuration elements into the existing configuration.

Merging configurations is performed according to the following rules. (The rules also apply when updating configuration data in an open configuration database, for example, the ephemeral database, but for simplicity the following discussion refers to the candidate configuration only.)

- A configuration element (hierarchy level or configuration object) that exists in the candidate configuration but not in the new configuration remains unchanged.
- A configuration element that exists in the new configuration but not in the candidate configuration is added to the candidate configuration.
- If a configuration element exists in both configurations, the following results occur:
  - If a child statement of the configuration element (represented by a child tag element) exists in the candidate configuration but not in the new configuration, it remains unchanged.
  - If a child statement exists in the new configuration but not in the candidate, it is added to the candidate configuration.
  - If a child statement exists in both configurations, the value in the new data replaces the value in the candidate configuration.

To explicitly specify that data be merged, the application includes the `<default-operation>` tag element with the value `merge` in the `<edit-config>` tag element.

```
<rpc>
    <edit-config>
        <default-operation>merge</default-operation>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
]]>]]>
```

Specifying the replace Data Mode

In the *replace* edit configuration mode, the new configuration data completely replaces the data in the candidate configuration or open configuration database. To specify that the data be replaced, the application includes the `<default-operation>` tag element with the value `replace` in the `<edit-config>` tag element.

```
<rpc>
    <edit-config>
        <default-operation>replace</default-operation>
    </edit-config>
</rpc>
]]>]]>
```

We recommend using the global replace mode only when you plan to completely overwrite the existing configuration with new configuration data. Furthermore, when the edit configuration mode is set to `replace`, we do not recommend using the `operation` attribute for individual configuration elements.

You can also replace individual configuration elements while merging or creating others. See "Replace Configuration Elements Using NETCONF" on page 314.

Specifying the none (no-change) Data Mode

In the `none` (*no-change*) edit configuration mode, changes to the configuration are ignored. This mode is useful when you are deleting elements, and it prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. For more information, see "Delete Configuration Elements Using NETCONF" on page 306.

To set the no-change edit configuration mode globally, the application includes the `<default-operation>` tag element with the value `none` in the `<edit-config>` tag element.

```
<rpc>
  <edit-config>
    <default-operation>none</default-operation>
  </edit-config>
</rpc>
```

NOTE: If the new configuration data includes a configuration element that is not in the existing configuration, the NETCONF server returns an error. We recommend using mode `none` only when removing configuration elements from the configuration. When creating or modifying elements, applications must use merge mode.

When you use the `<default-operation>` tag to globally set the edit configuration mode to `none` to indicate the no-change mode, you can still override this mode and specify a different edit configuration mode for individual elements by including the `operation` attribute in the element's tag. For example:

```
<rpc>
  <edit-config>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <services>
            <outbound-ssh>
              <client>
                <name>test</name>
                <device-id>test</device-id>
                <keep-alive>
                  <retry operation="merge">4</retry>
                  <timeout operation="merge">15</timeout>
                </keep-alive>
              </client>
            </outbound-ssh>
          </services>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.1R1 and 21.1R1-EVO	Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 21.1R1,when you set `<default-operation>` to `replace`, the device uses a `load update` operation instead of a `load override` operation to replace the configuration. In load update operations, the device notifies only the Junos processes that correspond to changed statements, thus minimizing possible disruptions to the network.

RELATED DOCUMENTATION

Delete Configuration Elements Using NETCONF | 306

Handle Errors While Editing the Candidate Configuration in a NETCONF Session

In a NETCONF session with a device running Junos OS, you can use NETCONF XML management protocol operations along with Junos XML or command-line interface (CLI) configuration statements to change the configuration on a routing, switching, or security platform. If the NETCONF server cannot incorporate the configuration data, the server returns the `<rpc-error>` tag element with information explaining the reason for the failure. By default, when the NETCONF server encounters an error while incorporating new configuration data into the candidate configuration, it halts the incorporation process. You can explicitly specify that the NETCONF server ignore errors or halt on error when incorporating new configuration data by including the `<error-option>` tag element.

A client application can explicitly specify that the NETCONF server stop incorporating new configuration data when it encounters an error. The application includes the `<error-option>` tag element with the value `stop-on-error` in the `<edit-config>` tag element.

```
<rpc>
    <edit-config>
        <error-option>stop-on-error</error-option>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
]]>]]>
```

Alternatively, the application can specify that the NETCONF server continue to incorporate new configuration data when it encounters an error. The application includes the `<error-option>` tag element with the value `ignore-error` in the `<edit-config>` tag element.

```
<rpc>
    <edit-config>
        <error-option>ignore-error</error-option>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
]]>]]>
```

The client application can include the optional `<test-option>` tag element described in the NETCONF specification. Regardless of the value provided, the NETCONF server for the Junos OS performs a basic syntax check on the configuration data in the `<edit-config>` tag element. When the `<test-option>` tag is included, NETCONF performs a complete syntactic and semantic validation in response to the `<commit>` and `<validate>` tag elements (that is, when the configuration is committed or explicitly checked), but not in response to the `<edit-config>` tag element.

RELATED DOCUMENTATION

Edit the Configuration Using NETCONF | 277

Verify the Candidate Configuration Syntax Using NETCONF | 322

Commit the Candidate Configuration Using NETCONF | 323

Upload and Format Configuration Data in a NETCONF Session | 279

Replace the Candidate Configuration Using NETCONF

IN THIS SECTION

A NETCONF client application can replace the entire candidate configuration or all data in the open configuration database, either with new data or by rolling back to a previous configuration or a rescue configuration.

NOTE: If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database, the device performs the requested operation on the open configuration database. Otherwise, the device performs the operation on the candidate configuration.

The following sections discuss how to replace configuration data in the candidate configuration or open configuration database. The client application must commit the configuration after replacing the data to make it the active configuration on the device.

Using <copy-config> to Replace the Configuration

NETCONF clients can use the `<copy-config>` operation to replace the entire candidate configuration or all data in the open configuration database. The `<target>` tag encloses the `<candidate/>` tag. The `<candidate/>` tag indicates that the new configuration data replaces the data in the open configuration database (if the client application issued the Junos XML protocol `<open-configuration>` operation prior to executing the `<copy-config>` operation), or if there is no open database, the data in the candidate configuration.

The `<source>` element specifies the source configuration datastore. The client application can specify the following sources:

- `<running/>`—Copy the active configuration to the target datastore.
- `<url>`—Copy the configuration data in the specified file to the target datastore.

  Include the `format` attribute to specify the format of the configuration data as `xml` (default) or `text`.

```
<rpc>
    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
            <running/>

         <!-- OR -->

            <url format="(xml | text)">
                <!-- location specifier for input file -->
            </url>
        </source>
    </copy-config>
</rpc>
```

Using <edit-config> to Replace the Configuration

A NETCONF client can also use the `<edit-config>` operation to replace the entire candidate configuration or all data in the open configuration database. The application includes the `<default-operation>` tag element and sets the edit configuration mode to `replace` as a global variable, as described in "Set the Edit Configuration Mode in a NETCONF Session" on page 287.

The `<target>` tag encloses the `<candidate/>` tag to indicate that the new configuration data replaces either the data in the open configuration database (if the client application issued the Junos XML protocol `<open-configuration>` operation prior to executing the `<edit-config>` operation), or if there is no open database, the data in the candidate configuration.

To specify the new configuration data, the application includes a `<config>` or `<config-text>` tag element that contains the data, or it includes a `<url>` tag element that specifies the file containing the data as discussed in "Upload and Format Configuration Data in a NETCONF Session" on page 279.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <default-operation>replace</default-operation>

    <!-- EITHER -->
        <config>
            <configuration>
                <!-- Junos XML configuration data -->
            </configuration>
        </config>
    <!-- OR -->
        <config-text>
            <configuration-text>
                <!-- configuration data in text format  -->
            </configuration-text>
        </config-text>
    <!-- OR -->
        <url>
            <!-- location specifier for file containing changes -->
        </url>

    </edit-config>
</rpc>
]]>]]>
```

Rolling Back to a Previously Committed Configuration

Junos devices store a copy of the most recently committed configuration and up to 49 previous configurations, depending on the platform. You can roll back to any of the stored configurations. This is useful when configuration changes cause undesirable results, and you want to revert back to a known working configuration. Rolling back the configuration is similar to the process for making configuration changes on the device, but instead of loading configuration data, you perform a rollback, which replaces the entire candidate configuration with a previously committed configuration.

Starting in Junos OS Release 18.1R1, a NETCONF application can execute the `<rollback-config>` RPC to replace either the candidate configuration or all data in the open configuration database with a previously committed configuration. To roll back the configuration, the application emits the `<rollback-config>` element with the `<index>` child element, which specifies the numerical index of the previous configuration to load. Valid values are 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49).

NOTE: NETCONF applications can also use the Junos XML protocol `<load-configuration>` operation with the `rollback` attribute to roll back the configuration.

For example, to load the configuration with a rollback index of 1, the client application emits the following RPC:

```
<rpc>
  <rollback-config>
    <index>1</index>
  </rollback-config>
</rpc>
]]>]]>
```

The NETCONF server indicates that the load operation was successful by returning the `<rollback-config-results>` and `<ok/>` elements in its RPC reply.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/18.1R1/junos">
  <rollback-config-results>
```

```
    <ok/>
  </rollback-config-results>
</rpc-reply>
]]>]]>
```

If the load operation is successful, the client application must commit the configuration to make it the active configuration on the device. If the server encounters an error while loading the rollback configuration, it returns an `<rpc-error>` element with information about the error.

Replacing the Candidate Configuration with the Rescue Configuration

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. You use the rescue configuration when you need to revert to a known configuration or as a last resort if the device configuration and the backup configuration files become damaged beyond repair. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration.

Starting in Junos OS Release 18.1R1, a NETCONF application can execute the `<rollback-config>` RPC to replace either the candidate configuration or all data in the open configuration database with the device's rescue configuration. To load the rescue configuration, the application emits the `<rollback-config>` element and `<rescue/>` child tag. The rescue configuration must exist on the device before you can load it.

NOTE: NETCONF applications can also use the Junos XML protocol `<load-configuration>` operation with the `rescue` attribute to load the rescue configuration.

For example, to load the rescue configuration, the client application emits the following RPC:

```
<rpc>
  <rollback-config>
    <rescue/>
  </rollback-config>
</rpc>
]]>]]>
```

The NETCONF server indicates that the load operation was successful by returning the `<rollback-config-results>` and `<ok/>` elements in its RPC reply.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/18.1R1/junos">
  <rollback-config-results>
    <ok/>
  </rollback-config-results>
</rpc-reply>
]]>]]>
```

If the load operation is successful, the client application must commit the configuration to make it the active configuration on the device. If the rescue configuration does not exist or the server encounters another error while loading the configuration data, it returns an `<rpc-error>` element with information about the error.

RELATED DOCUMENTATION

Set the Edit Configuration Mode in a NETCONF Session | **287**

Replace Configuration Elements Using NETCONF | **314**

Upload and Format Configuration Data in a NETCONF Session | **279**

<copy-config> | **186**

<edit-config> | **191**

Roll Back Uncommitted Changes in the Candidate Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, the client application can roll back the candidate configuration to the current running configuration, which removes any uncommitted changes from the candidate configuration. This operation is equivalent to the CLI configuration mode `rollback 0` command.

To roll back the candidate configuration to the current running configuration, enclose the `<discard-changes>` tag within the `<rpc>` element.

```
<rpc>
    <discard-changes/>
</rpc>
]]>]]>
```

After you issue the `</discard-changes>` tag, the NETCONF server indicates that it successfully discarded the changes by returning the `<ok/>` tag.

RELATED DOCUMENTATION

Replace the Candidate Configuration Using NETCONF | 293

Retrieve a Previous (Rollback) Configuration Using NETCONF | 425

<discard-changes/> | 190

Delete the Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, the `<delete-config>` tag element enables you to delete all configuration data in the current candidate configuration or in the open configuration database. Exercise caution when issuing the `<delete-config>` tag element. If you commit an empty candidate configuration, the device will go offline.

To delete the candidate configuration or all data in the open configuration database, insert the `<delete-config>` tag element in the `<rpc>` element. The `<target>` tag encloses the `<candidate/>` tag, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing a `<delete-config>` operation, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

```
<rpc>
  <delete-config>
    <target>
      <candidate/>
    </target>
  </delete-config>
</rpc>
```

WARNING: If you take the device offline, you will need to access the device through the console port on the device. From this console, you can access the CLI and perform a rollback to a suitable configuration. For more information about the console port, see the hardware manual for your specific device.

RELATED DOCUMENTATION

Delete Configuration Elements Using NETCONF | 306

Replace the Candidate Configuration Using NETCONF | 293

Roll Back Uncommitted Changes in the Candidate Configuration Using NETCONF | 298
<delete-config> | 189

Change Individual Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, a client application can change individual configuration elements in the existing configuration by using the `<edit-config>` tag element. By default, the NETCONF server merges new configuration data into the existing configuration. However, a client application can also replace, create, or delete individual configuration elements (hierarchy levels or configuration objects). The same basic tag elements are emitted for all operations: `<config>`, `<config-text>`, or `<url>` tag sub-elements within the `<edit-config>` tag element.

Within the `<edit-config>` element, the `<target>` element encloses the `<candidate/>` tag, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<edit-config>` operation, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>

    <!-- EITHER -->
        <config>
            <configuration>
                <!-- tag elements representing the configuration elements to change -->
            </configuration>
        </config>
    <!-- OR -->
        <config-text>
            <configuration-text>
                <!-- configuration data in text format -->
            </configuration-text>
        </config-text>
    <!-- OR -->
        <url>
            <!-- location specifier for file containing changes -->
        </url>
    </edit-config>
</rpc>
]]>]]>
```

The application includes the configuration data within the `<config>` or `<config-text>` tag elements or in the file specified by the `<url>` tag element. To define a configuration element, the application includes the tag elements representing all levels of the configuration hierarchy from the root down to the immediate parent level for the element. To represent the element, the application includes its container tag element. The child tags included within the container element depend on the operation.

For more information about the tag elements that represent configuration statements, see "Map Configuration Statements to Junos XML Tag Elements" on page 25. For information about the tag elements for a specific configuration element, see the *Junos XML API Configuration Developer Reference*.

The NETCONF server indicates that it changed the configuration in the requested way by enclosing the `<ok/>` tag in the `<rpc-reply>` tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Create Configuration Elements Using NETCONF | 304

Delete Configuration Elements Using NETCONF | 306

Merge Configuration Elements Using NETCONF | 301

Replace Configuration Elements Using NETCONF | 314

Merge Configuration Elements Using NETCONF

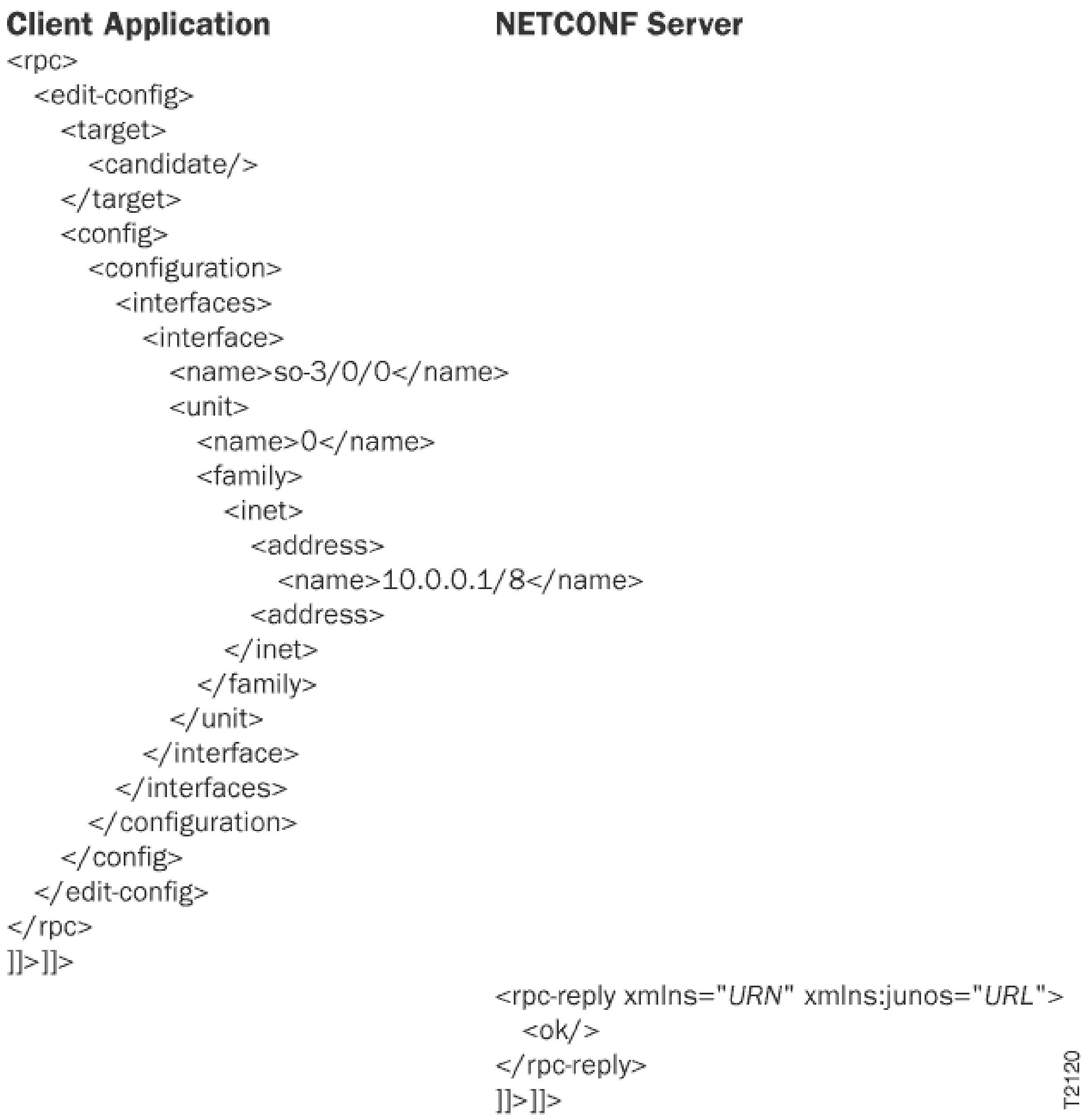
In a NETCONF session with a device running Junos OS, to merge configuration elements, including hierarchy levels or configuration objects, into the existing configuration in the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol `<open-configuration>` operation prior to executing the `<edit-config>` operation), a client application emits the basic tag elements described in "Change Individual Configuration Elements Using NETCONF" on page 300.

To represent each element to merge in (either within the `<config>` or `<config-text>` tag elements or in the file specified by the `<url>` tag element), the application includes the tag elements representing its parent hierarchy levels and its container tag element, as described in "Change Individual Configuration Elements Using NETCONF" on page 300. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and the tag element for each child to add or for which to set a different value. In the following, the identifier tag element is called `<name>`:

```
<configuration>
    <!-- opening tags for each parent of the element -->
        <element>
            <name>identifier</name>
            <!-- - child tag elements to add or change -->
        </element>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server merges in the new configuration element according to the rules specified in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. As described in that section, the application can explicitly specify merge mode by including the `<default-operation>` tag element with the value `merge` in the `<edit-config>` tag element.

The following example shows how to merge information for a new interface called so-3/0/0 into the `[edit interfaces]` hierarchy level in the candidate configuration:



Client Application

```
<rpc>
   <edit-config>
      <target>
         <candidate/>
      </target>
      <config>
         <configuration>
            <interfaces>
               <interface>
                  <name>so-3/0/0</name>
                  <unit>
                     <name>0</name>
                     <family>
                        <inet>
                           <address>
                              <name>10.0.0.1/8</name>
                           <address>
                        </inet>
                     </family>
                  </unit>
               </interface>
            </interfaces>
         </configuration>
      </config>
   </edit-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
   <ok/>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Change Individual Configuration Elements Using NETCONF | 300

Create Configuration Elements Using NETCONF | 304

Delete Configuration Elements Using NETCONF | 306

Replace Configuration Elements Using NETCONF | 314

Set the Edit Configuration Mode in a NETCONF Session | 287

Create Configuration Elements Using NETCONF

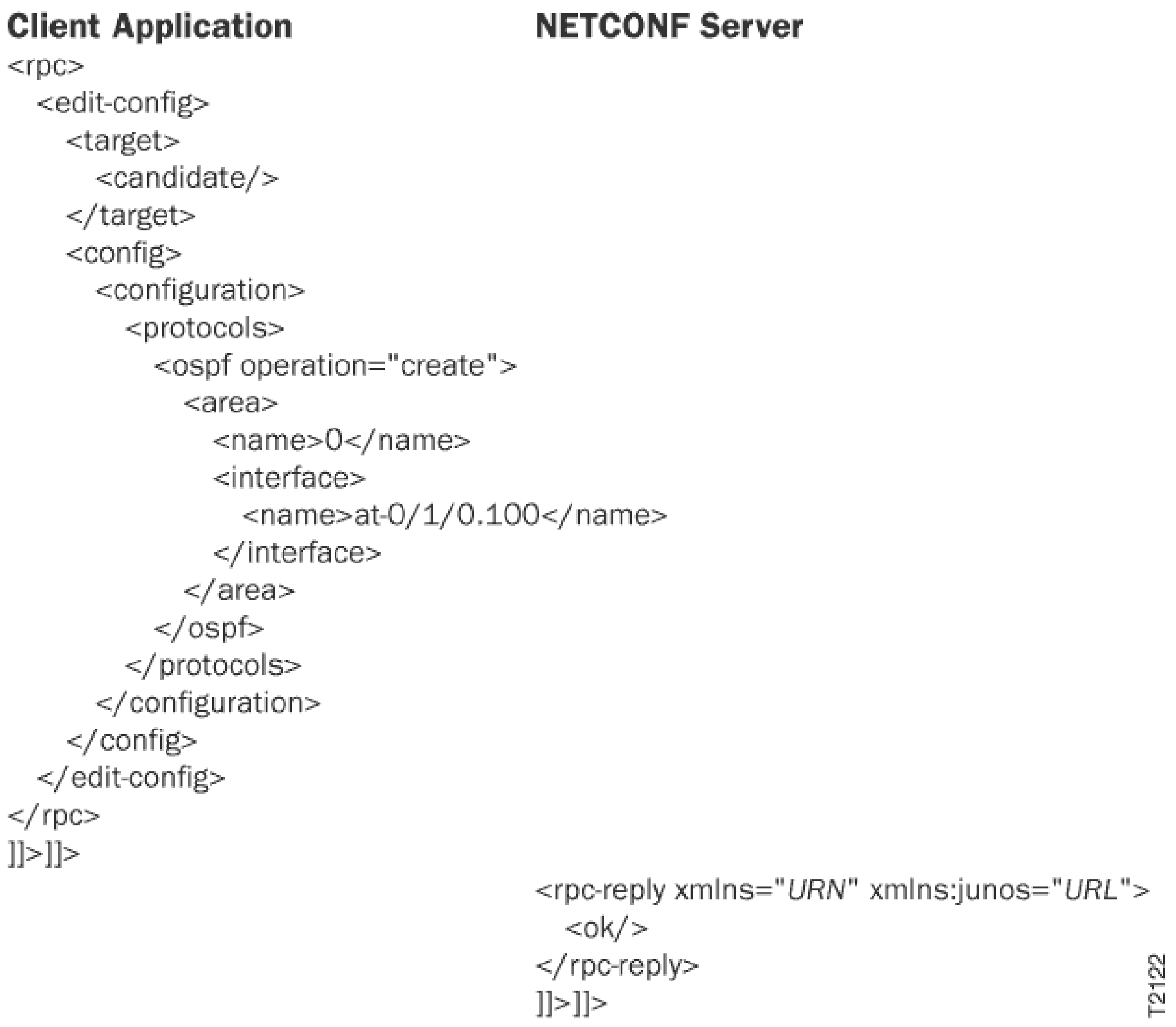
In a NETCONF session with a device running Junos OS, to create configuration elements, including hierarchy levels or configuration objects, that do not already exist in the target configuration, which can be either the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol `<open-configuration>` operation prior to executing the `<edit-config>` operation), a client application emits the basic tag elements described in "Change Individual Configuration Elements Using NETCONF" on page 300.

To represent each configuration element being created (either within the `<config>` or `<config-text>` tag elements or in the file specified by the `<url>` tag element), the application emits the tag elements representing its parent hierarchy levels and its container tag element, as described in "Change Individual Configuration Elements Using NETCONF" on page 300. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and all child tag elements (with values, if appropriate) that are being defined for the element. In the following, the identifier tag element is called `<name>`. The application includes the `operation="create"` attribute in the opening container tag:

```
<configuration>
    <!-- opening tags for each parent of the element -->
        <element operation="create">
            <name>identifier</name>  <!-- if element has an identifier -->
            <!-- other child tag elements -->
        </element>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server adds the new element to the target configuration only if there is no existing element with that name (for a hierarchy level) or with the same identifiers (for a configuration object).

The following example shows how to enable OSPF on a device if it is not already configured:



```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <protocols>
          <ospf operation="create">
            <area>
              <name>0</name>
              <interface>
                <name>at-0/1/0.100</name>
              </interface>
            </area>
          </ospf>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

- Change Individual Configuration Elements Using NETCONF | **300**
- Delete Configuration Elements Using NETCONF | **306**
- Merge Configuration Elements Using NETCONF | **301**
- Replace Configuration Elements Using NETCONF | **314**
- Set the Edit Configuration Mode in a NETCONF Session | **287**

Delete Configuration Elements Using NETCONF

SUMMARY

Delete configuration elements

IN THIS SECTION

You can use NETCONF to delete configuration elements, including hierarchy levels or configuration objects, from the Junos configuration. You can delete objects from the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol `<open-configuration>` operation prior to executing the `<edit-config>` operation).

To delete an element, a client application emits the basic tag elements described in "Change Individual Configuration Elements Using NETCONF" on page 300. It also emits the `<default-operation>` tag element with the value `none` to change the default mode to `no-change`.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <default-operation>none</default-operation>

   <!-- EITHER -->
        <config>
            <configuration>
                <!-- tag elements representing the configuration elements to delete -->
            </configuration>
        </config>
   <!-- OR -->
        <url>
            <!-- location specifier for file containing elements to delete -->
        </url>

    </edit-config>
</rpc>
]]>]]>
```

In no-change mode, existing configuration elements remain unchanged unless the corresponding element in the new configuration has the `operation="delete"` attribute or `operation="remove"` attribute in its opening tag. This mode prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. We recommend that client applications only perform deletion operations when using no-change mode. When merging, replacing, or creating configuration elements, client applications use merge mode.

The `delete` and `remove` attributes both instruct the NETCONF server to delete the specified configuration element in the target configuration. However, the NETCONF server behaves differently for each attribute when the specified configuration element does not exist. If you use the `remove` attribute and the element does not exist, the server silently ignores the request. If you use the `delete` attribute and the element does not exist, the server returns an `<rpc-error>` element with an `<error-tag>` value of `data-missing`. We recommend that you use the `remove` attribute when you want to delete an element but are unsure if the element exists.

To represent each configuration element being deleted (either within the `<config>` tag element or in the file named by the `<url>` tag element), the application emits the tag elements representing its parent hierarchy levels, as described in "Change Individual Configuration Elements Using NETCONF" on page 300. The placement of the `operation="delete"` attribute or `operation="remove"` attribute depends on the element type, as described in the following sections.

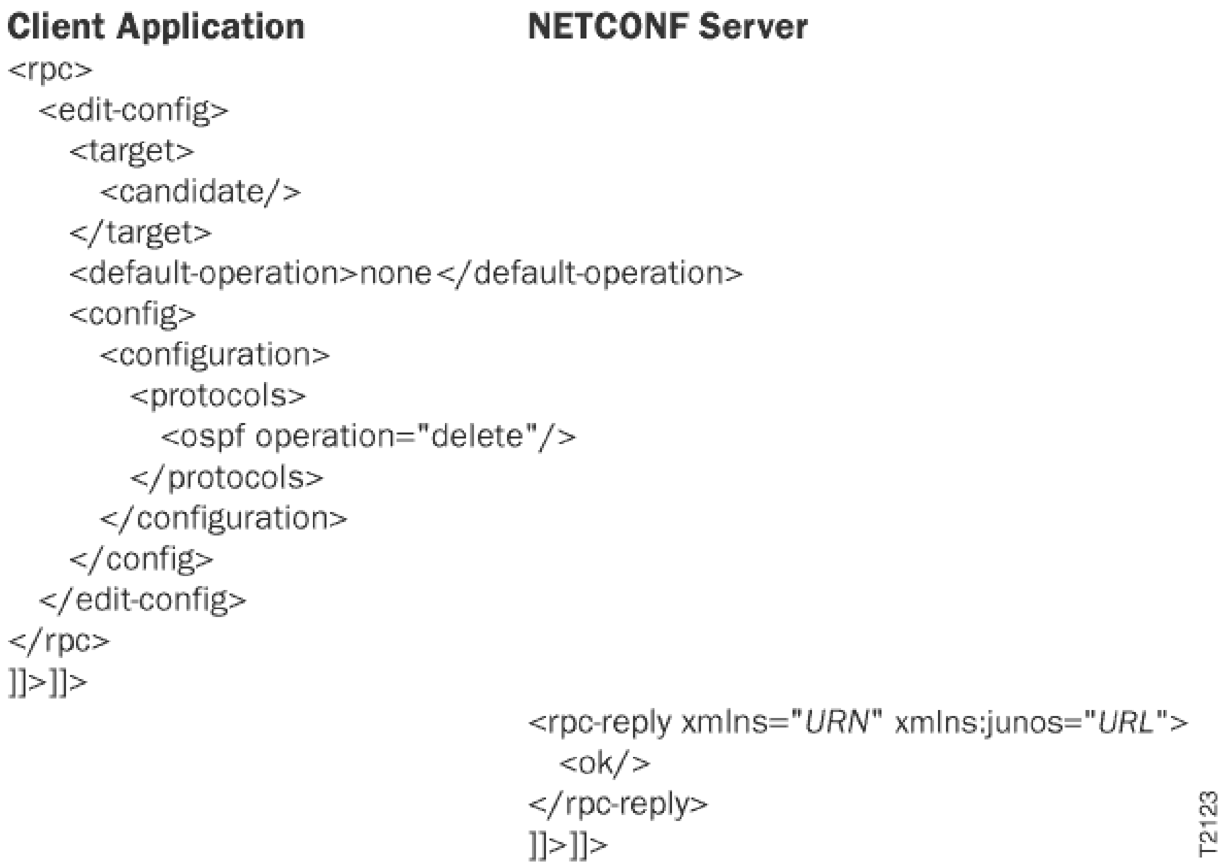
Deleting a Hierarchy Level or Container Object

To delete a hierarchy level and all of its children (or a container object that has children but no identifier), a client application includes the `operation="delete"` attribute or the `operation="remove"` attribute in the empty tag that represents the level.

```
<configuration>
    <!-- opening tags for each parent level -->
        <level-to-delete operation="(delete | remove)"/>
    <!-- closing tags for each parent level -->
</configuration>
```

We recommend that the application set the default mode to no-change by including the `<default-operation>` tag element with the value `none`, as described in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. For more information about hierarchy levels and container objects, see "Map Configuration Statements to Junos XML Tag Elements" on page 25.

The following example shows how to remove the `[edit protocols ospf]` hierarchy level of the candidate configuration:



```
Client Application                  NETCONF Server
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <protocols>
          <ospf operation="delete"/>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
                                    <rpc-reply xmlns="URN" xmlns:junos="URL">
                                      <ok/>
                                    </rpc-reply>
                                    ]]>]]>
```

Deleting a Configuration Object That Has an Identifier

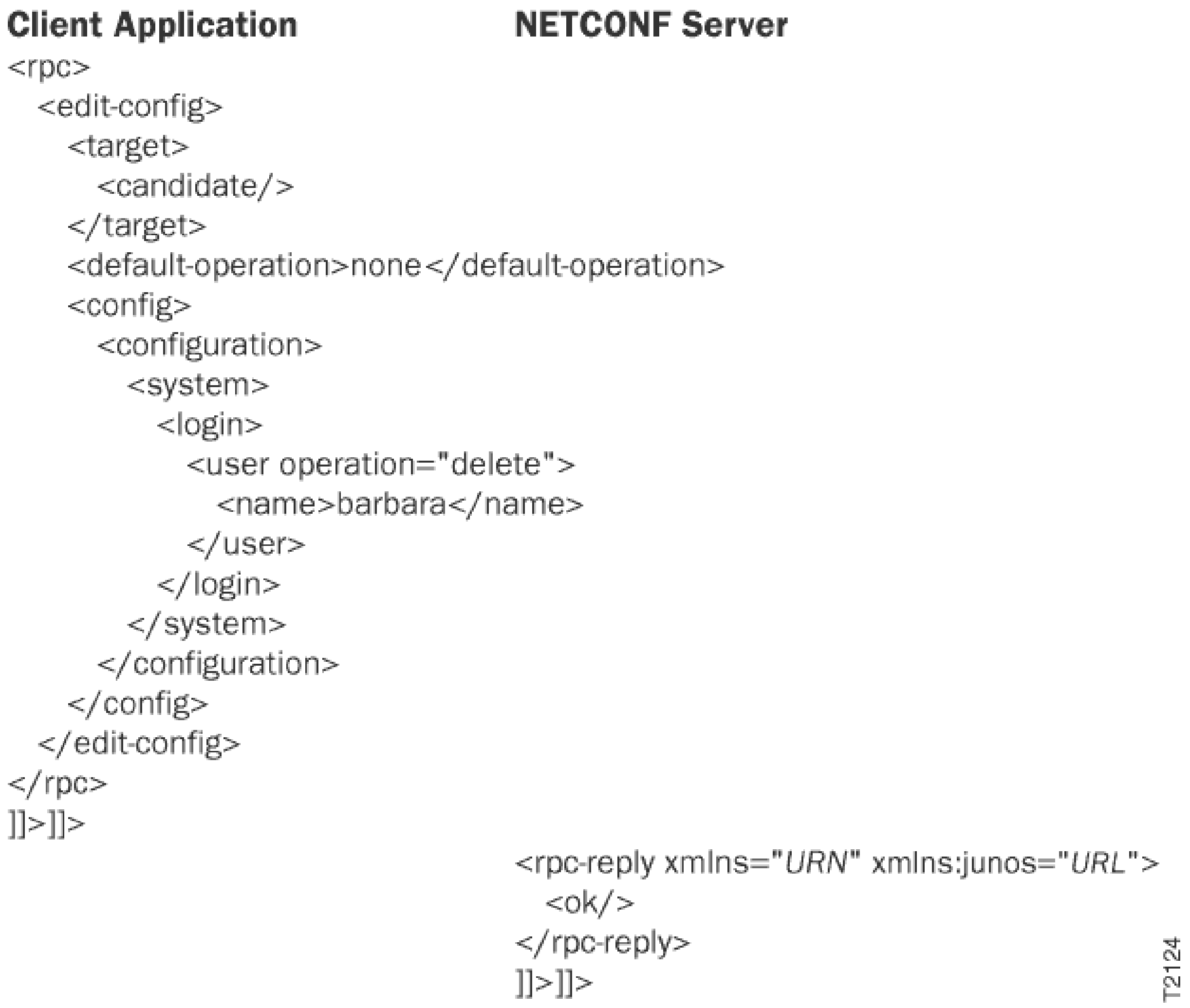
To delete a configuration object that has an identifier, a client application includes the `operation="delete"` attribute or the `operation="remove"` attribute in the container tag element for the object. Inside the container tag element, it includes the identifier tag element only, not any tag elements that represent other characteristics. In the following, the identifier tag element is called `<name>`:

```
<configuration>
    <!-- opening tags for each parent of the object -->
        <object operation="(delete | remove)">
            <name>identifier</name>
        </object>
    <!-- closing tags for each parent of the object -->
</configuration>
```

NOTE: The `delete` or `remove` attribute appears in the opening container tag, not in the identifier tag element. You include the identifier tag element to delete the specified object, not the entire hierarchy level represented by the container tag.

We recommend that the application set the default mode to no-change by including the `<default-operation>` tag element with the value `none`, as described in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. For more information about identifiers, see "Map Configuration Statements to Junos XML Tag Elements" on page 25.

The following example shows how to remove the user object `barbara` from the `[edit system login user]` hierarchy level in the candidate configuration:



```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <login>
            <user operation="delete">
              <name>barbara</name>
            </user>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

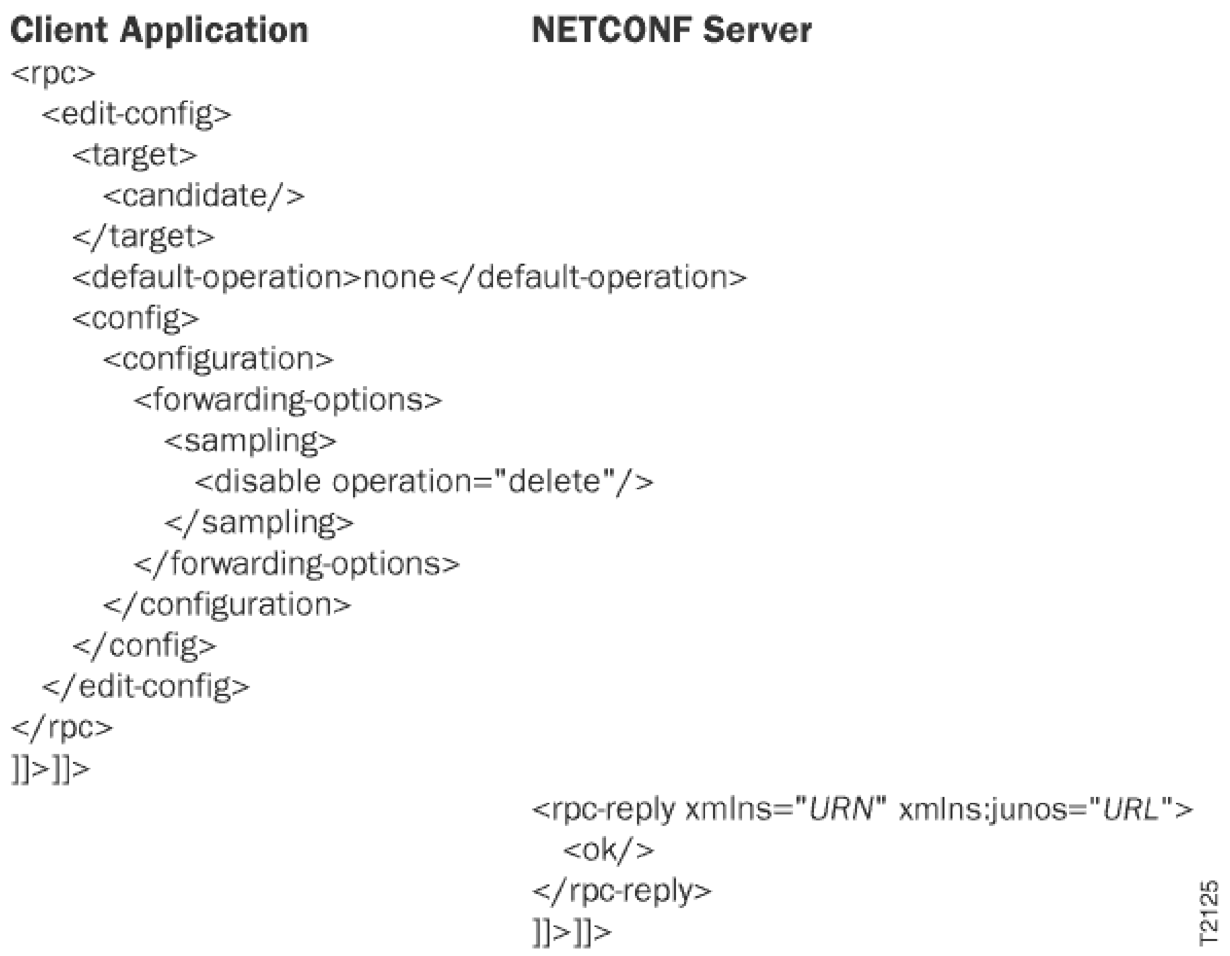
Deleting a Single-Value or Fixed-Form Option from a Configuration Object

To delete either a fixed-form option or an option that takes just one value from a configuration object, a client application includes the `operation="delete"` attribute or the `operation="remove"` attribute in the tag element for the option. In the following example, the identifier tag element for the object is called `<name>`. (For information about deleting an option that can take multiple values, see "Deleting Values from a Multi-value Option of a Configuration Object" on page 311.)

```
<configuration>
    <!-- opening tags for each parent of the object -->
        <object>
            <name>identifier</name>  <!-- if object has an identifier -->
            <option1 operation="(delete | remove)">
            <option2 operation="(delete | remove)">
            <!-- tag elements for other options to delete -->
        </object>
    <!-- closing tags for each parent of the object -->
</configuration>
```

We recommend that the application set the default mode to no-change by including the `<default-operation>` tag element with the value `none`, as described in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. For more information about options, see "Map Configuration Statements to Junos XML Tag Elements" on page 25.

The following example shows how to remove the fixed-form option `disable` at the `[edit forwarding-options sampling]` hierarchy level:



```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <forwarding-options>
          <sampling>
            <disable operation="delete"/>
          </sampling>
        </forwarding-options>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

Deleting Values from a Multi-value Option of a Configuration Object

As described in "Map Configuration Statements to Junos XML Tag Elements" on page 25, some Junos OS configuration objects are leaf statements that have multiple values. In the formatted ASCII CLI representation, the values are enclosed in square brackets following the name of the object:

```
object [value1 value2 value3 ...];
```

The Junos XML representation does not use a parent tag for the object, but instead uses a separate instance of the object tag element for each value. In the following example, the identifier tag element is called `<name>`:

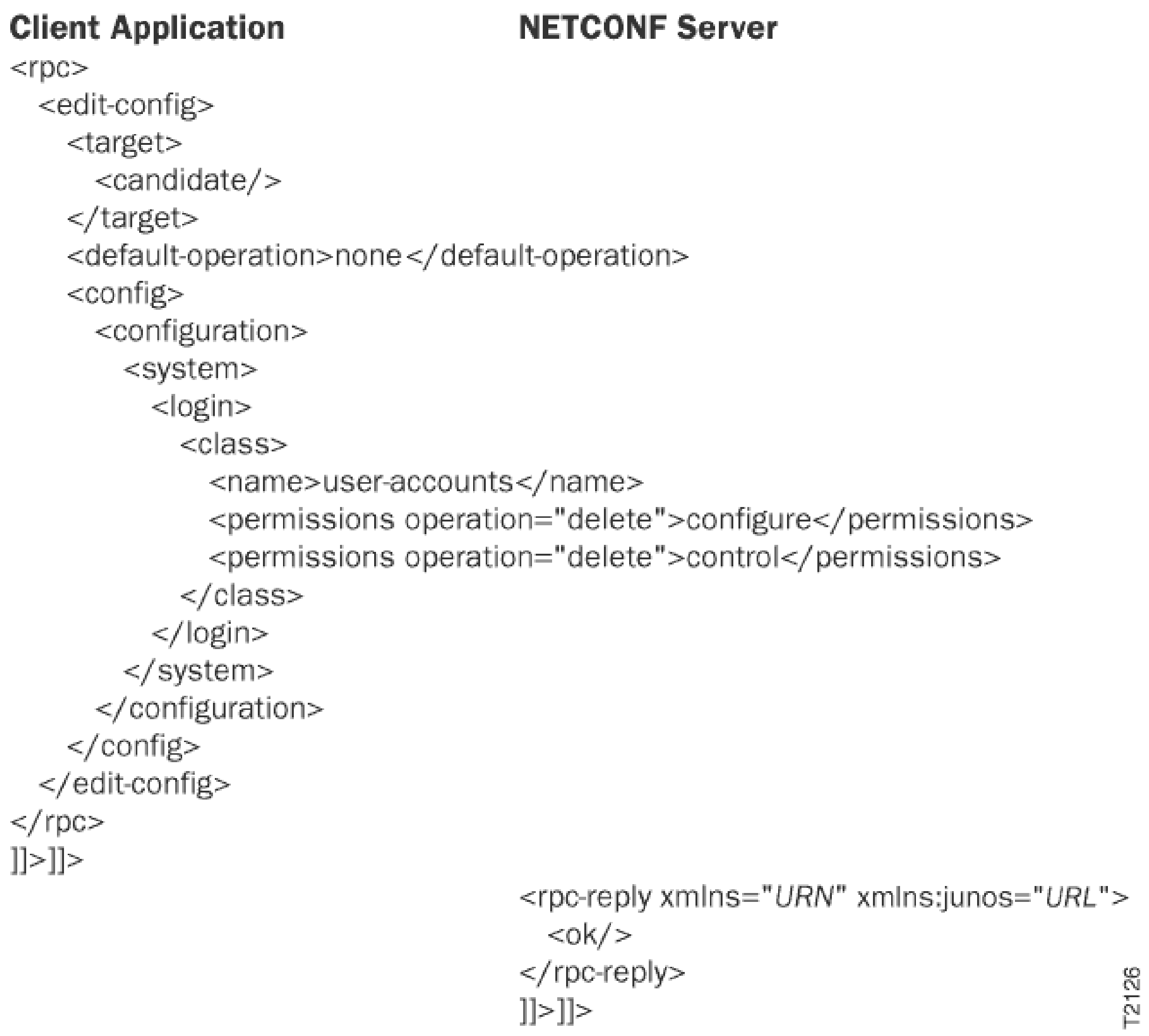
```
<parent-object>
    <name>identifier</name>
    <object>value1</object>
    <object>value2</object>
    <object>value3</object>
</parent-object>
```

To remove one or more values for such an object, a client application includes the `operation="delete"` attribute or the `operation="remove"` attribute in the opening tag for each value. It does not include tag elements that represent values to be retained. In the following example, the identifier tag element is called `<name>`:

```
<configuration>
    <!-- opening tags for each parent of the parent object -->
        <parent-object>
            <name>identifier</name>
            <object operation="(delete | remove)">value1</object>
            <object operation="(delete | remove)">value2</object>
        </parent-object>
    <!-- closing tags for each parent of the parent object -->
</configuration>
```

We recommend that the application set the default mode to no-change by including the `<default-operation>` tag element with the value `none`, as described in "Set the Edit Configuration Mode in a NETCONF Session" on page 287. For more information about leaf statements with multiple values, see "Map Configuration Statements to Junos XML Tag Elements" on page 25.

The following example shows how to remove two of the permissions granted to the `user-accounts` login class:



```
Client Application                    NETCONF Server
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <login>
            <class>
              <name>user-accounts</name>
              <permissions operation="delete">configure</permissions>
              <permissions operation="delete">control</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
                                      <rpc-reply xmlns="URN" xmlns:junos="URL">
                                        <ok/>
                                      </rpc-reply>
                                      ]]>]]>
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.1R1 and 23.1R1-EVO	Starting in Junos OS Release 23.1R1 and Junos OS Evolved Release 23.1R1, the NETCONF server `<rpc-error>` response is changed when `<edit-config>` uses the `operation="delete"` operation to delete a configuration element that is absent in the target configuration. The error severity is error instead of warning, and the `<rpc-error>` element includes the `<error-tag>data-missing</error-tag>` and `<error-type>application</error-type>` elements.

RELATED DOCUMENTATION

Change Individual Configuration Elements Using NETCONF | 300

Delete the Configuration Using NETCONF | 299

Create Configuration Elements Using NETCONF | 304

Merge Configuration Elements Using NETCONF | 301

Replace Configuration Elements Using NETCONF | 314

Set the Edit Configuration Mode in a NETCONF Session | 287

Replace Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, to replace configuration elements, including hierarchy levels or configuration objects, in the candidate configuration, a client application emits the basic tag elements described in "Change Individual Configuration Elements Using NETCONF" on page 300.

To represent the new definition for each configuration element being replaced (either within the `<config>` or `<config-text>` tag elements or in the file specified by the `<url>` tag element), the application emits the tag elements representing its parent hierarchy levels and its container tag element, as described in "Change Individual Configuration Elements Using NETCONF" on page 300. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and all child tag elements (with values, if appropriate) that are being defined for the new version of the element. In the following example, the identifier tag element is called `<name>`. The application includes the `operation="replace"` attribute in the opening container tag:

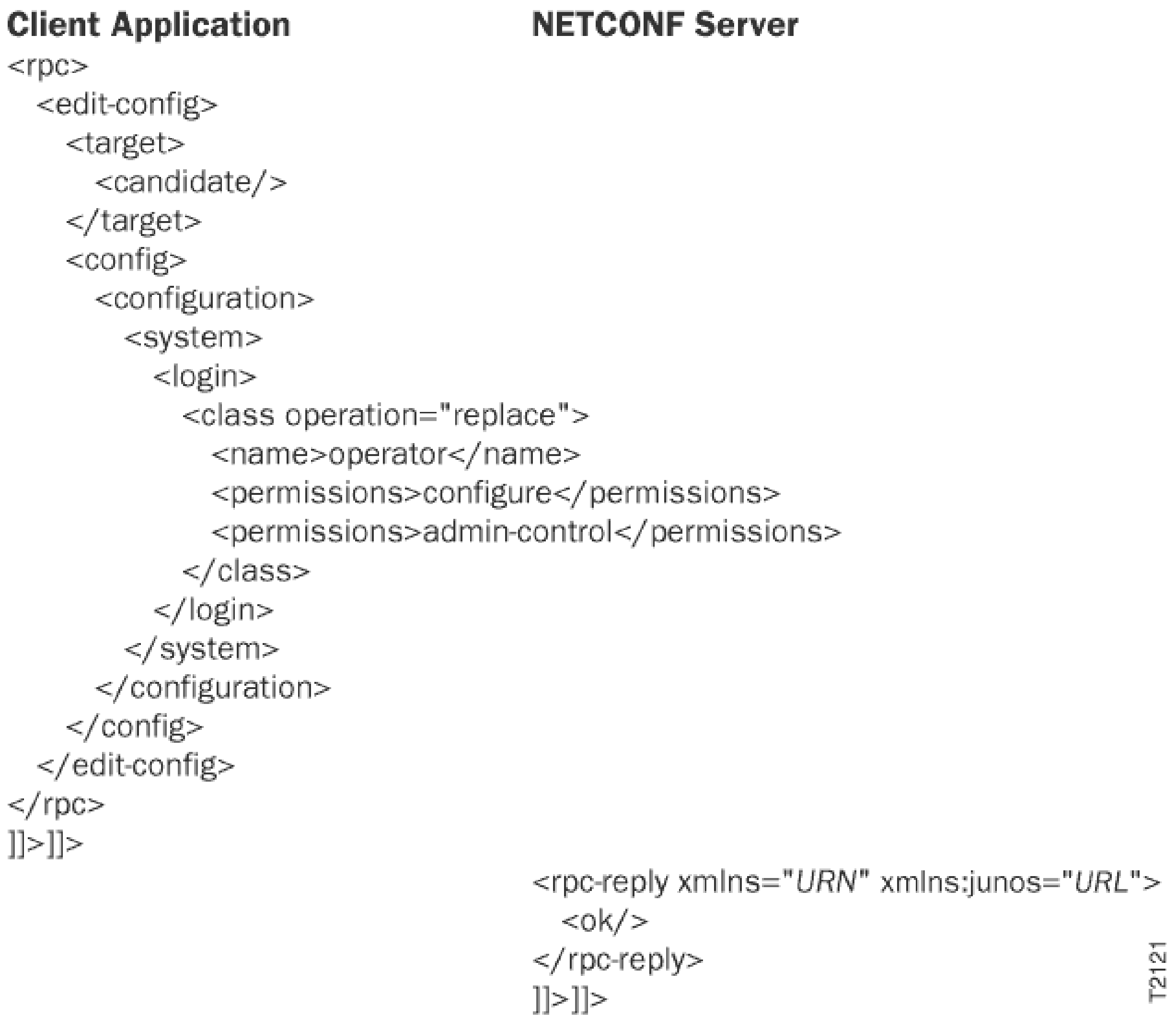
```
<configuration>
    <!-- opening tags for each parent of the element -->
        <container-tag operation="replace">
            <name>identifier</name>
            <!-- other child tag elements -->
        </container-tag>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server removes the existing element that has the specified identifiers and inserts the new element.

NOTE: The `operation="replace"` attribute is not supported when loading configuration data into the ephemeral configuration database.

The application can also replace all objects in the configuration in one operation. For instructions, see "Replace the Candidate Configuration Using NETCONF" on page 293.

The following example shows how to grant new permissions for the object named `operator` at the `[edit system login class]` hierarchy level.



```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <login>
            <class operation="replace">
              <name>operator</name>
              <permissions>configure</permissions>
              <permissions>admin-control</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Change Individual Configuration Elements Using NETCONF | 300

Create Configuration Elements Using NETCONF | 304

Replace Patterns in Configuration Data Using the NETCONF or Junos XML Protocol

SUMMARY

NETCONF and Junos XML protocol client applications can replace variables and identifiers in the configuration when performing a `<load-configuration>` operation.

IN THIS SECTION

A NETCONF or Junos XML protocol client application can replace variables and identifiers in the configuration of devices running Junos OS or devices running Junos OS Evolved. For example, you might need to replace all occurrences of an interface name when a PIC is moved to another slot in the router.

To replace a pattern, a client application uses the `<load-configuration>` operation with the `replace-pattern` attribute. The `replace-pattern` attribute replaces the existing pattern with the new pattern. The scope of the replacement can be global or at a specified hierarchy or object level in the configuration. The functionality of the `replace-pattern` attribute is identical to that of the `replace pattern` configuration mode command in the Junos OS CLI.

NOTE: To use the replace pattern operations, you must use Junos XML elements for the configuration data format.

To replace a pattern, a client application emits the `<rpc>` and `<load-configuration>` elements and includes the basic Junos XML tag elements described in *Create, Modify, or Delete Configuration Elements Using the Junos XML Protocol*. At the hierarchy or object level where you want to replace the pattern, include the following attributes:

- `replace-pattern`—Pattern to replace.

- `with`—Replacement pattern.
- `upto`—(Optional) Number of occurrences to replace. If you omit this attribute or set it to zero, the device replaces all instances of the pattern within the specified scope.

The placement of the attributes within the configuration determines the scope of the replacement as described in the following sections.

Replace Patterns Globally Within the Configuration

To replace a pattern globally throughout the candidate configuration or open configuration database, include the `replace-pattern` and `with` attributes in the opening `<configuration>` tag. You can optionally include the `up-to` attribute to replace only a specified number of occurrences.

```
<rpc>
   <load-configuration>
      <configuration replace-pattern="pattern1" with="pattern2" [upto="n"]>
      </configuration>
   </load-configuration>
</rpc>
```

For example, the following RPC replaces all instances of 172.17.1.5 with 172.16.1.1:

```
<rpc>
   <load-configuration>
      <configuration replace-pattern="172.17.1.5" with="172.16.1.1"/>
      </configuration>
   </load-configuration>
</rpc>
```

After executing the RPC, you can compare the updated candidate configuration to the active configuration to verify the pattern replacement. You must commit the configuration for the changes to take effect.

```
<rpc>
   <get-configuration compare="rollback" rollback="0" format="text">
   </get-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/24.4R1/junos">
<configuration-information>
<configuration-output>
```

```
[edit groups global system ntp]
-    boot-server 172.17.1.5;
+    boot-server 172.16.1.1;
[edit groups global system ntp]
+     server 172.16.1.1;
-     server 172.17.1.5;
</configuration-output>
</configuration-information>
</rpc-reply>
```

Replace Patterns Within a Hierarchy Level or Container Object That Has No Identifier

A client application can replace a pattern under a specific hierarchy level including all of its children (or a container object that has children but no identifier). To replace the pattern within a specific hierarchy level, a client application includes the `replace-pattern` and `with` attributes in the empty tag that represents the hierarchy level or container object.

```
<rpc>
   <load-configuration>
      <configuration>
         <!-- opening tag for each parent element -->
            <level-or-object replace-pattern="pattern1" with="pattern2" [upto="n"]/>
         <!-- closing tag for each parent element -->
      </configuration>
   </load-configuration>
</rpc>
```

The following RPC replaces instances of fe-0/0/1 with ge-1/0/1 at the `[edit interfaces]` hierarchy level:

```
<rpc>
   <load-configuration>
      <configuration>
         <interfaces replace-pattern="fe-0/0/1" with="ge-1/0/1"/>
      </configuration>
    </load-configuration>
</rpc>
```

Compare the updated candidate configuration to the active configuration to verify the pattern replacement. For example:

```
<rpc>
   <get-configuration compare="rollback" rollback="0" format="text">
   </get-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/24.4R1/junos">
<configuration-information>
<configuration-output>
[edit interfaces]
-   fe-0/0/1 {
-       unit 0 {
-           family inet {
-               address 10.0.1.1/27;
-           }
-       }
-   }
+   ge-1/0/1 {
+       unit 0 {
+           family inet {
+               address 10.0.1.1/27;
+           }
+       }
+   }
</configuration-output>
</configuration-information>
</rpc-reply>
```

Replace Patterns for a Configuration Object That Has an Identifier

To replace a pattern for a configuration object that has an identifier, a client application includes the `replace-pattern` and `with` attributes in the opening tag for the object. Within the container tag, the application also includes the identifier element for that object. In the following example, the identifier tag is `<name>`:

```
<rpc>
   <load-configuration>
      <configuration>
         <!-- opening tag for each parent element -->
```

```
            <container-tag replace-pattern="pattern1" with="pattern2" [upto="n"]>
               <name>identifier</name>
            </container-tag>
         <!-- closing tag for each parent element -->
      </configuration>
   </load-configuration>
</rpc>
```

The following RPC replaces instances of 4.5 with 4.1, but only for the fe-0/0/2 interface under the `[edit interfaces]` hierarchy:

```
<rpc>
   <load-configuration>
      <configuration>
         <interfaces>
            <interface replace-pattern="4.5" with="4.1">
               <name>fe-0/0/2</name>
            </interface>
         </interfaces>
      </configuration>
   </load-configuration>
</rpc>
```

You can compare the updated candidate configuration to the active configuration to verify the pattern replacement. For example:

```
<rpc>
   <get-configuration compare="rollback" rollback="0" format="text">
   </get-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/24.4R1/junos">
<configuration-information>
<configuration-output>
[edit interfaces fe-0/0/2 unit 0 family inet]
+       address 10.0.4.1/30;
-       address 10.0.4.5/30;
</configuration-output>
</configuration-information>
```

RELATED DOCUMENTATION

replace-pattern

Modifying the Configuration for a Device

Modifying the Configuration for a Device

replace

CHAPTER 11

Commit the Configuration Using NETCONF

IN THIS CHAPTER

Verify the Candidate Configuration Syntax Using NETCONF

In a NETCONF session with a device running Junos OS, during the process of committing the candidate configuration or a private copy, the NETCONF server confirms that the configuration is syntactically correct. If the syntax check fails, the server does not commit the candidate configuration. To avoid the potential complications of such a failure, it often makes sense to confirm the correctness of the candidate configuration before actually committing it.

In a NETCONF session with a device running Junos OS, to verify the syntax of the candidate configuration, a client application includes the `<validate>` and `<source>` tag elements and the `<candidate/>` tag in an `<rpc>` tag element:

```
<rpc>
    <validate>
        <source>
            <candidate/>
        </source>
    </validate>
</rpc>
]]>]]>
```

The NETCONF server confirms that the candidate configuration syntax is valid by returning the `<ok/>` tag in the `<rpc-reply>` tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the candidate configuration syntax is not valid, the server returns the `<rpc-reply>` element and `<rpc-error>` child element, which explains the reason for the error.

RELATED DOCUMENTATION

Commit the Candidate Configuration Using NETCONF | 323

Commit the Candidate Configuration Only After Confirmation Using NETCONF | 325

Commit the Candidate Configuration Using NETCONF

When you commit the candidate configuration on a device running Junos OS, it becomes the active configuration on the routing, switching, or security platform. For more detailed information about commit operations, including a discussion of the interaction among different variants of the operation, see the CLI User Guide.

In a NETCONF session with a device running Junos OS, to commit the candidate configuration, a client application encloses the `<commit/>` tag in an `<rpc>` tag element.

```
<rpc>
    <commit/>
</rpc>
]]>]]>
```

We recommend that the client application lock the candidate configuration before modifying it and emit the `<commit/>` tag while the configuration is still locked. This process avoids inadvertently committing changes made by other users or applications. After committing the configuration, the application must unlock it in order for other users and applications to make changes.

The NETCONF server confirms that the commit operation was successful by returning the `<ok/>` tag in the `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the commit operation fails, the server returns the `<rpc-reply>` element and `<rpc-error>` child element, which explains the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

You can configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level to enforce certain behaviors by the NETCONF server, which includes changes in the NETCONF server's response to `<commit>` operations. Table 16 on page 324 describes the changes in RFC-compliant sessions.

Table 16: Commit RPC Response Differences in RFC-Compliant Sessions

Commit RPC Response	Default Response	RFC-Compliant Session Response
A successful `<commit>` operation returns a response with warnings.	The NETCONF server returns an `<ok/>` element and can also return one or more `<rpc-error>` elements with a severity level of warning.	Starting in Junos OS Release 17.4R3, 18.2R2, 18.3R2, and 18.4R1, the NETCONF server returns an `<ok/>` element but omits any warnings. In Junos OS Release 21.2R1 and later, the warnings are also redirected to the system log file.
A `<commit>` operation response returns an `<rpc-error>` element that includes a `<source-daemon>` element.	The NETCONF server response emits the `<source-daemon>` element as a child of `<rpc-error>`.	Starting in Junos OS Release 21.2R1, the NETCONF server response emits the `<source-daemon>` element as a child of `<error-info>`.
A `<commit>` operation response includes a `<commit-results>` element.	The NETCONF server includes the `<commit-results>` XML subtree in addition to an `<ok/>` element or `<rpc-error>` child element.	If you also configure the `flatten-commit-results` statement at the `[edit system services netconf]` hierarchy level, the NETCONF server suppresses the `<commit-results>` XML subtree and only emits an `<ok/>` or `<rpc-error>` element in its response.

RELATED DOCUMENTATION

Commit the Candidate Configuration Only After Confirmation Using NETCONF | 325

Commit the Candidate Configuration Only After Confirmation Using NETCONF

When you commit the candidate configuration on a device running Junos OS, it becomes the active configuration on the routing, switching, or security platform. For more detailed information about commit operations, including a discussion of the interaction among different variants of the operation, see the CLI User Guide

When you commit the candidate configuration, you can require an explicit confirmation for the commit to become permanent. The confirmed commit operation is useful for verifying that a configuration change works correctly and does not prevent management access to the device. If the change prevents access or causes other errors, the automatic rollback to the previous configuration restores access after the rollback deadline passes. If the commit is not confirmed within the specified amount of time, which is 600 seconds (10 minutes) by default, the device automatically loads and commits (rolls back to) the previously committed configuration.

In a NETCONF session with a device running Junos OS, to commit the candidate configuration but require an explicit confirmation for the commit to become permanent, a client application encloses the empty `<confirmed/>` tag in the `<commit>` and `<rpc>` tag elements.

```
<rpc>
    <commit>
        <confirmed/>
    </commit>
</rpc>
]]>]]>
```

To specify a number of seconds for the rollback deadline that is different from the default value of 600 seconds, the application includes the `<confirm-timeout>` tag element, and specifies the number of seconds for the delay, in the range from 1 through 4,294,967,295 seconds.

```
<rpc>
    <commit>
        <confirmed/>
        <confirm-timeout>rollback-delay</confirm-timeout>
    </commit>
```

```
</rpc>
]]>]]>
```

NOTE: You cannot perform a confirmed commit operation on an instance of the ephemeral configuration database.

In either case, the NETCONF server confirms that it committed the candidate configuration temporarily by returning the `<ok/>` tag in the `<rpc-reply>`.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot commit the candidate configuration, the `<rpc-reply>` element instead encloses an `<rpc-error>` element explaining the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

To delay the rollback to a time later than the current rollback deadline, the client application emits the `<confirmed/>` tag in a `<commit>` tag element again before the deadline passes. Optionally, it includes the `<confirm-timeout>` element to specify how long to delay the next rollback; omit that tag element to delay the rollback by the default of 600 seconds (10 minutes). The client application can delay the rollback indefinitely by emitting the `<confirmed/>` tag repeatedly in this way.

To commit the configuration permanently, the client application emits the `<commit/>` tag enclosed in an `<rpc>` tag element before the rollback deadline passes. The rollback is canceled and the candidate configuration is committed immediately, as described in "Commit the Candidate Configuration Using NETCONF" on page 323. If the candidate configuration is still the same as the temporarily committed configuration, this effectively recommits the temporarily committed configuration.

If another application uses the `<kill-session/>` tag element to terminate this application’s session while a confirmed commit is pending (this application has committed changes but not yet confirmed them), the NETCONF server that is servicing this session restores the configuration to its state before the confirmed commit instruction was issued.

The following example shows how to commit the candidate configuration with a rollback deadline of 300 seconds.

Client Application

```
<rpc>
  <commit>
```

```
    <confirmed/>
    <confirm-timeout>300</confirm-timeout>
  </commit>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Commit the Candidate Configuration Using NETCONF | **323**

CHAPTER 12

Ephemeral Configuration Database

IN THIS CHAPTER

Understanding the Ephemeral Configuration Database

IN THIS SECTION

The *ephemeral database* is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS and Junos OS Evolved. The ephemeral database enables Juniper Extension Toolkit (JET) applications and NETCONF and Junos XML management protocol client applications to concurrently load and commit configuration changes to a device and with significantly greater throughput than when committing data to the candidate configuration database.

Use Feature Explorer to confirm platform and release support for specific features.

The following sections discuss the different aspects of the ephemeral configuration database.

Benefits of the Ephemeral Configuration Database

- Enables multiple client applications to concurrently configure a device by loading and committing data to separate instances of the ephemeral database
- Enables fast provisioning and rapid configuration changes in dynamic environments that require fast commit times

Ephemeral Configuration Database Overview

When managing Junos devices, the recommended and most common method to configure the device is to modify and commit the candidate configuration, which corresponds to a persistent (static) configuration database. The standard commit operation handles configuration groups, macros, and commit scripts; performs commit checks to validate the configuration's syntax and semantics; and stores copies of the committed configurations. The standard commit model is robust because it prevents configuration errors and it enables you to roll back to a previously committed configuration. However, in some cases, the commit operation can consume a significant amount of time and device resources.

JET applications and NETCONF and Junos XML protocol client applications can also configure the ephemeral database. The ephemeral database is an alternate configuration database that provides a configuration layer separate from both the candidate configuration database and the configuration layers of other client applications. The ephemeral commit model enables Junos devices to commit and merge changes from multiple clients and execute the commits with significantly greater throughput than when committing data to the candidate configuration database. Thus, the ephemeral database is advantageous in dynamic environments where fast provisioning and rapid configuration changes are required, such as in large data centers.

A commit operation on the ephemeral database requires less time than the same operation on the static database because the ephemeral database is not subject to the same validation required in the static database. As a result, the ephemeral commit model provides better performance than the standard commit model but at the expense of some of the more robust features present in the standard model. The ephemeral commit model has the following restrictions:

- Configuration data syntax is validated, but configuration data semantics are not validated.
- Certain configuration statements are not supported as described in *Unsupported Configuration Statements in the Ephemeral Configuration Database*.
- Configuration groups and interface ranges are not processed.
- Macros, commit scripts, and translation scripts are not processed.

- Previous versions of the ephemeral configuration are not archived.
- Standard show commands do not display ephemeral configuration data in the output.
- Ephemeral configuration data does not persist when you:
  - Install a package that requires rebuilding the Junos schema, for example, an OpenConfig or YANG package.
  - Perform a software upgrade or a unified in-service software upgrade (ISSU).
  - Reboot or power cycle the device.

CAUTION: We strongly recommend that you exercise caution when using the ephemeral configuration database. Committing invalid configuration data can corrupt the ephemeral database, which can cause Junos processes to restart or stop responding and result in disruption to the system or network.

Junos devices validate the syntax but not the semantics, or constraints, of the configuration data committed to the ephemeral database. For example, if the configuration references an undefined routing policy, the configuration might be syntactically correct, but it would be semantically incorrect. The standard commit model generates a commit error in this case, but the ephemeral commit model does not. Therefore, it is imperative to validate all configuration data before committing it to the ephemeral database. If you commit configuration data that is invalid or results in undesirable network disruption, you must remove the problematic data from the database. You can delete the data, or if necessary, you can reboot the device, which deletes the configuration data in all instances of the ephemeral configuration database.

NOTE: The ephemeral configuration database stores internal version information in addition to configuration data. As a result, the size of the ephemeral configuration database is always larger than the static configuration database for the same configuration data, and most operations on the ephemeral database, whether additions, modifications, or deletions, increase the size of the database.

NOTE: When you use the ephemeral configuration database, commit operations on the static configuration database might take longer, because the device must perform additional operations to merge the static and ephemeral configuration data.

Ephemeral Database Instances

Junos devices provide a default ephemeral database instance, which is automatically enabled. You can also enable multiple user-defined instances of the ephemeral configuration database. JET applications and NETCONF and Junos XML protocol client applications can concurrently load and commit data to separate instances of the ephemeral database. The active device configuration is a merged view of the static and ephemeral configuration databases.

Ephemeral database instances are useful when multiple client applications need to simultaneously update a device configuration. For example, two or more SDN controllers might need to simultaneously push configuration data to the same device. In the standard commit model, one controller might have an exclusive lock on the candidate configuration, thereby preventing the other controller from modifying it. By using separate ephemeral instances, the controllers can deploy the changes at the same time.

NOTE: Applications can simultaneously load and commit data to different ephemeral database instances in addition to the static configuration database. However, the device processes the commits sequentially. As a result, the commit to a specific database might be delayed, depending on the processing order.

The Junos processes read the configuration data from both the static configuration database and the ephemeral configuration database. When one or more ephemeral database instances are in use and there is conflicting data, statements in a database with a higher priority override the statements in a database with a lower priority. The database priority, from highest to lowest, is as follows:

1. Statements in a user-defined instance of the ephemeral configuration database.

   If the device uses multiple user-defined ephemeral instances, it determines the priority by the order in which the instances are configured at the `[edit system configuration-database ephemeral]` hierarchy level, running from highest to lowest priority.

2. Statements in the default ephemeral database instance.

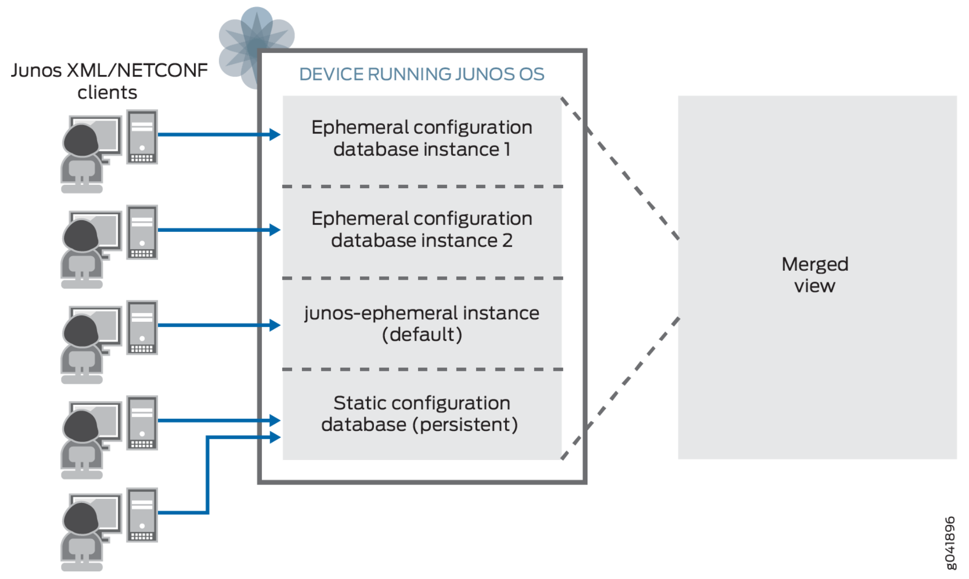
3. Statements in the static configuration database.

Consider the following configuration:

```
system {
    configuration-database {
        ephemeral {
            instance 1;
            instance 2;
        }
    }
}
```

Figure 6 on page 332 illustrates the priority of the ephemeral database instances and the static (committed) configuration database. In this example, ephemeral database instance 1 has the highest priority, followed by ephemeral database instance 2, then the default ephemeral database instance, and finally the static configuration database.

Figure 6: Ephemeral Database Instances


Junos XML/NETCONF clients
DEVICE RUNNING JUNOS OS
Ephemeral configuration database instance 1
Ephemeral configuration database instance 2
junos-ephemeral instance (default)
Static configuration database (persistent)
Merged view
g041896

Ephemeral Database General Commit Model

JET applications and NETCONF and Junos XML protocol client applications can modify the ephemeral configuration database. JET applications must send configuration requests as pairs of load and commit operations. NETCONF and Junos XML protocol client applications can perform multiple load operations before executing a commit operation.

CAUTION: You must validate all configuration data before loading it into the ephemeral database and committing it on the device. Committing invalid configuration data can cause Junos processes to restart or stop responding and result in disruption to the system or network.

Client applications can simultaneously load and commit data to different ephemeral database instances. Commits issued at the same time for different ephemeral instances are queued and processed serially by the device. If a client disconnects from a session, the device discards any uncommitted configuration changes in the ephemeral instance. However, configuration data that has already been committed to the ephemeral instance by that client is unaffected.

When you commit an ephemeral instance, the system validates the syntax, but not the semantics, of the ephemeral configuration data. When the commit is complete, the device notifies the affected system processes. The processes read the updated configuration and merge the ephemeral data into the active configuration according to the rules of prioritization described in "Ephemeral Database Instances" on page 330. The active device configuration is a merged view of the static and ephemeral configuration databases.

NOTE: The ephemeral database's commit time will be slightly longer on devices running Junos OS Evolved than on devices running Junos OS because of the architectural differences between the two operating systems.

For detailed information about committing and synchronizing instances of the ephemeral configuration database, see *Commit and Synchronize Ephemeral Configuration Data Using the NETCONF or Junos XML Protocol*.

Using the Ephemeral Database with High Availability Features

High availability refers to the hardware and software components that provide redundancy and reliability for network communications. You should consider certain behaviors and caveats before using the ephemeral database on systems that use high availability features. High availability features include redundant Routing Engines, graceful Routing Engine switchover (GRES), nonstop active routing (NSR), and interchassis redundancy for MX Series routers or EX Series switches using Virtual Chassis. The following sections describe these behaviors and outline how the different ephemeral database commit synchronize models can affect these behaviors.

Understanding Ephemeral Database Commit Synchronize Models

The ephemeral configuration database has two models for synchronizing ephemeral configuration data across Routing Engines or Virtual Chassis members during a commit synchronize operation:

- Asynchronous
- Synchronous

Starting with Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the ephemeral database uses the synchronous model by default and devices that enable GRES or NSR must use the synchronous model. However, you can still use either model to synchronize ephemeral data on devices that do not enable GRES or NSR. In earlier releases, the asynchronous model is the default.

Asynchronous Model

When the ephemeral database uses the asynchronous commit model, the primary Routing Engine or Virtual Chassis primary device commits the configuration and then notifies the backup device. The requesting Routing Engine does not wait for the other Routing Engine to first synchronize and commit the configuration. Devices that use high availability features require that the primary and backup Routing Engines are synchronized in case of a failover. However, there can be situations in which an asynchronous commit synchronize operation can be interrupted and fail to synchronize the ephemeral configuration to the other Routing Engine.

Synchronous Model

The synchronous commit model is similar to the model used by the static configuration database. Synchronous commit operations are slower than asynchronous commit operations, but they provide better assurance that the ephemeral configuration is synchronized across Routing Engines or Virtual Chassis members. Thus, the synchronous commit model enables you to use the ephemeral database with greater reliability on devices that use high availability features.

In a dual Routing-Engine or MX Series Virtual Chassis environment, the synchronous commit model works as follows:

1. The primary Routing Engine or Virtual Chassis primary device starts its commit operation for the ephemeral instance.

2. At a given point during the commit operation, the device initiates a commit on the backup Routing Engine or Virtual Chassis backup device.

3. If the other Routing Engine successfully commits the configuration, then the primary Routing Engine continues its commit operation. If the commit fails on the other Routing Engine, then the primary Routing Engine also fails the commit.

When an EX Series Virtual Chassis uses the synchronous commit model, the member switch in the primary Routing Engine role first initiates the commit operation on the other members simultaneously. Because an EX Series Virtual Chassis can have many members, the primary switch then proceeds with its commit operation, even if the commit fails on another member.

NOTE: As is the case for the static configuration database, even with the synchronous commit synchronize model, there can be rare circumstances in which the device commits an updated ephemeral configuration on the backup Routing Engine but fails to complete the commit on the primary Routing Engine resulting in the configurations being out of synchronization.

Failover Synchronization

Devices running Junos OS Release 20.2R1 or later and devices running Junos OS Evolved also support failover configuration synchronization for the ephemeral database. If you configure failover synchronization, then when the backup Routing Engine synchronizes with the primary Routing Engine, for example, when it is newly inserted, brought back online, or during a change in role, it synchronizes both its static and ephemeral configuration databases. To enable failover synchronization, configure the `commit synchronize` statement at the `[edit system]` hierarchy level in the static configuration database.

NOTE: For failover synchronization, the backup Routing Engine or the MX Virtual Chassis backup device only synchronizes the ephemeral configuration database with the primary device if both the backup device and the primary device are running the same software version.

Both commit synchronize operations and failover synchronize operations synchronize the ephemeral configuration data to the other Routing Engine using a load update operation instead of a load override operation. By using a load update operation, the device only needs to notify the Junos processes that correspond to changed statements during the update, which minimizes possible disruptions to the network.

Redundant Routing Engines

Dual Routing Engine systems support configuring the ephemeral database. However, the ephemeral commit model does not automatically synchronize ephemeral configuration data to the backup Routing Engine during a commit operation. You can synchronize the data in an ephemeral instance on a per-commit or per-session basis. You can also configure an ephemeral instance to automatically synchronize its data every time you commit the instance. For more information, see *Commit and Synchronize Ephemeral Configuration Data Using the NETCONF or Junos XML Protocol*.

NOTE: Multichassis environments do not support synchronizing the ephemeral configuration database to the other Routing Engines.

When a client application commits data in an ephemeral instance and synchronizes it to the backup Routing Engine, the device synchronizes the ephemeral data using the configured commit synchronize model. Dual Routing Engine devices also support failover configuration synchronization for the ephemeral database. For more information, see "Understanding Ephemeral Database Commit Synchronize Models" on page 333.

Graceful Routing Engine Switchover (GRES)

GRES enables a device with redundant Routing Engines to continue forwarding packets, even if the primary Routing Engine fails. GRES requires that the primary and backup Routing Engines synchronize the configuration and certain state information before a switchover occurs.

We recommend that you use the synchronous commit synchronize model on devices that enable GRES. Moreover, starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES must use the synchronous model. Synchronous commit operations are slower than asynchronous commit operations, but they provide better assurance that the ephemeral configuration is synchronized between Routing Engines.

NOTE: Dual Routing Engine devices running Junos OS Evolved enable GRES by default.

Although supported in certain releases, we do *not* recommend using the asynchronous commit synchronize model on devices that enable GRES. If you use the asynchronous model, in certain circumstances, the ephemeral database might not be synchronized between the primary and backup Routing Engines when a switchover occurs. For example, the backup and primary Routing Engines might not synchronize the ephemeral database if the commit synchronize operation is interrupted by a sudden power outage. If you use the asynchronous commit model on a GRES-enabled device, you must explicitly configure the device to synchronize ephemeral configuration data to the backup Routing Engine. To enable synchronization, configure the `allow-commit-synchronize-with-gres` statement at the `[edit system configuration-database ephemeral]` hierarchy level in the static configuration database.

Nonstop Active Routing (NSR)

Nonstop active routing (NSR) enables the transparent switchover of the Routing Engines in the event that one of the Routing Engines goes down. We recommend that you use the synchronous commit synchronize model on devices that enable NSR. Moreover, starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable NSR must use the synchronous model. Synchronous commit operations are slower than asynchronous commit operations, but they provide better assurance that the ephemeral configuration is synchronized between Routing Engines.

Although supported in certain releases, we do *not* recommend using the asynchronous commit synchronize model on devices that enable NSR, because it comes with certain caveats. In a deployment with dual Routing Engines, a commit synchronize operation on an ephemeral instance on the primary Routing Engine results in an asynchronous commit on the backup Routing Engine. If the device notifies the routing protocol process (rpd) in the process of updating the configuration, it could result in an undesirable behavior of the system due to the asynchronous nature of the commit on the backup Routing Engine. In Junos OS Release 21.1R1 and later, the device synchronizes the ephemeral instance to the backup Routing Engine using a load update operation, so it only notifies processes corresponding to statements that are changed.

NOTE: Applications utilizing the ephemeral database are only impacted in this NSR situation if they interact with the routing protocol process. For example, the SmartWall Threat Defense Director (SmartWall TDD) would not be impacted in this case, because it only interacts with the firewall process (dfwd) through the ephemeral database.

MX Series Virtual Chassis

MX Series Virtual Chassis support configuring the ephemeral database. You can configure and commit an ephemeral instance only on the primary Routing Engine of the Virtual Chassis primary device.

An MX Series Virtual Chassis does not automatically synchronize any ephemeral configuration data during a commit operation. As with dual Routing Engine systems, you can synchronize the data in an ephemeral instance on a per-commit or per-session basis. You can also configure an ephemeral instance to automatically synchronize its data every time you commit the instance. The device synchronizes the ephemeral data only from the primary Routing Engine on the primary device to the primary Routing Engine on the backup device.

NOTE: MX Series Virtual Chassis do not, under any circumstance, synchronize ephemeral configuration data from the primary Routing Engine to the backup Routing Engine on the respective Virtual Chassis member.

MX Series Virtual Chassis must have GRES configured. If you configure the ephemeral database to use the synchronous commit synchronize model (recommended), the device synchronizes the ephemeral instance to the other Routing Engine when you request a commit synchronize operation. However, if the ephemeral database uses the asynchronous commit synchronize model, you must explicitly configure the `allow-commit-synchronize-with-gres` statement in the static configuration database to enable synchronization. See "Understanding Ephemeral Database Commit Synchronize Models" on page 333 for more information about the ephemeral database commit models.

NOTE: Starting with Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES must use the synchronous commit synchronize model.

When you commit and synchronize an ephemeral instance on an MX Series Virtual Chassis that uses the asynchronous commit synchronize model:

1. The Virtual Chassis primary device validates the configuration syntax and commits the ephemeral instance on its primary Routing Engine.

2. If the commit is successful, the primary device notifies the backup device to synchronize the ephemeral instance.

3. The backup device commits the ephemeral instance on its primary Routing Engine only. If the commit operation fails, the backup device logs a message in the system log file but does not notify the primary device.

When you commit and synchronize an ephemeral instance on an MX Series Virtual Chassis that is configured to use the synchronous commit synchronize model, which is the recommended method:

1. The Virtual Chassis primary device starts its commit of the ephemeral instance on its primary Routing Engine.
2. At a given point in its commit operation, the primary device initiates a commit on the backup device's primary Routing Engine.
3. If the backup device successfully commits the configuration, then the primary device proceeds with its commit operation. If the backup device fails to commit the configuration, then the primary device also fails the commit.

As outlined, when you use the asynchronous commit synchronize model for the ephemeral database, the commit can succeed on the primary device but fail on the backup device. When you use the synchronous commit synchronize model, the commit either succeeds or fails for both primary Routing Engines, except in rare circumstances.

MX Series Virtual Chassis support failover configuration synchronization for the ephemeral database. To configure failover configuration synchronization, include the `commit synchronize` statement at the `[edit system]` hierarchy level in the static configuration database. After you configure the statement, the primary Routing Engine on the Virtual Chassis backup device synchronizes both its static and ephemeral configuration databases when it synchronizes with the primary Routing Engine on the Virtual Chassis primary device, for example, after it restarts.

NOTE: For failover synchronization, the MX Virtual Chassis backup device only synchronizes the ephemeral configuration database with the primary device if both devices are running the same software version.

EX Series Virtual Chassis

EX Series Virtual Chassis support the ephemeral configuration database. You can only configure and commit an ephemeral instance on the member switch in the primary Routing Engine role. Starting in Junos OS Release 23.4R1, you can synchronize the ephemeral database across EX Series Virtual Chassis members.

An EX Series Virtual Chassis does not automatically synchronize any ephemeral configuration data during a commit operation. You can synchronize the data in an ephemeral instance on a per-commit or per-session basis. You can also configure an ephemeral instance to automatically synchronize its data every time you commit the instance.

You can configure GRES on an EX Series Virtual Chassis to enable the Virtual Chassis to continue forwarding packets if the primary Routing Engine fails. If you configure the ephemeral database to use the synchronous commit synchronize model (recommended), the device synchronizes the ephemeral instance to the other members when you request a commit synchronize operation. However, if the ephemeral database uses the asynchronous commit synchronize model and GRES is configured, you must explicitly configure the `allow-commit-synchronize-with-gres` statement in the static configuration database to enable synchronization.

NOTE: Starting with Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES must use the synchronous commit synchronize model.

When you commit and synchronize an ephemeral instance on an EX Series Virtual Chassis that uses the asynchronous commit synchronize model:

1. The member switch in the primary Routing Engine role validates the configuration syntax and commits the ephemeral instance.

2. If the commit is successful, the primary device notifies the `commit-syncd` process, which initiates the commit on each member switch in turn.

3. Each member switch commits the ephemeral instance. If the commit operation fails on any member, it does not affect the commit operation on the other members.

When you commit and synchronize an ephemeral instance on an EX Series Virtual Chassis that is configured to use the synchronous commit synchronize model, which is the recommended method:

1. The member switch in the primary Routing Engine role initiates the commit on all member switches simultaneously.

2. Each member switch commits the ephemeral instance and notifies the primary switch. If the commit operation fails on any member, it does not affect the commit operation on the other members.

3. After receiving responses from all member switches, the primary switch commits the ephemeral instance.

As outlined, in the asynchronous model, the primary switch relies on the `commit-syncd` process to initiate the commits on each member switch sequentially. If the `commit-syncd` process fails for any reason, then some commits might not be initiated. In the synchronous commit model, the primary switch initiates the commit on all member switches directly and in parallel. Thus, the synchronous commit model is generally more reliable than the asynchronous commit model. In either case, if the commit fails on one member, it does not impact or prevent the commit on the other members.

Additionally, in the synchronous commit model, the primary switch displays the commit progress for each member as the commit occurs. In the asynchronous model, the commits occur in the background, so in this case, only the primary device logs the commit results.

Ephemeral Database Best Practices

The ephemeral configuration database enables multiple applications to make rapid configuration changes simultaneously. Because the ephemeral configuration database does not use the same safeguards as the static configuration database, you should carefully consider how you use the ephemeral database. We recommend following these best practices to optimize performance and avoid potential issues when you use the ephemeral configuration database.

Regulate Commit Frequency

The ephemeral database is designed for faster commits. However, committing too frequently can cause problems if the applications that consume the configuration can't keep pace with the rate of commit operations. Therefore, we recommend that you commit the next set of changes only after the device's operational state reflects the changes from the previous commit.

For example, if you execute frequent, rapid commits, the device could overwrite certain configuration data that it stores in external files before a Junos process reads the previous update. If a Junos process misses an important update, the device or network could exhibit unpredictable behavior.

Ensure Data Integrity

Junos devices do not validate configuration data semantics when you commit data to an ephemeral database. Therefore, you must take additional steps before loading and committing the configuration to ensure data integrity. We recommend that you always:

- Validate configuration data before loading it in the database
- Consolidate related configuration statements into a single database

You should validate all configuration data before loading it into an ephemeral database. We recommend that you pre-validate your configuration data using a static database, which validates both syntax and semantics.

Additionally, you should always load related configuration data into a single database. Adding related or dependent configuration data in the same database helps ensure that the device can detect and process related statements during a commit operation. For example, if you define a firewall filter in the static configuration database or in an ephemeral configuration database, then you should configure the application of the filter to an interface in the same configuration database.

By contrast, suppose you configure some statements in the static database but you configure related or dependent statements in an ephemeral database. When you commit the static database, the system validates the data only within that database. The system might not identify the dependent configuration in the ephemeral database, which can cause the validation, and thus the commit, to fail.

Consolidate Scaled Configurations

We recommend that you load scaled configurations into a single ephemeral database instance, rather than distributing them across multiple databases. A scaled configuration might include, for example, large lists of:

- Policy options
- Prefix lists
- VLANs
- Firewall filters

When you restrict a top-level configuration hierarchy to a single database, internal optimizations enable Junos processes to consume the configuration more efficiently. Alternatively, if you spread the configuration across multiple databases, Junos processes must parse a merged view of the configuration, which generally requires more resources and processing time.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the default commit synchronize model is `synchronous`. In earlier releases, the default is `asynchronous`.
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES or NSR must use the synchronous commit synchronize model.
21.1R1	Starting in Junos OS Release 21.1R1, the device synchronizes the ephemeral database instance to the backup Routing Engine using a load update operation instead of a load override operation. As a result, Junos OS only notifies processes corresponding to the changed statements.
20.2R1	Starting in Junos OS Release 20.2R1, when you configure the `commit synchronize` statement at the `[edit system]` hierarchy level in the static configuration database and the backup Routing Engine synchronizes with the primary Routing Engine, for example, when it is newly inserted, brought back online, or during a change in role, it synchronizes both its static and ephemeral configuration databases. In earlier releases, the device synchronizes only the static configuration database.
20.2R1	Starting in Junos OS Release 20.2R1, if an ephemeral database instance is already synchronized between the primary and backup Routing Engines, and you update the ephemeral instance on the primary Routing Engine, Junos OS only notifies the processes corresponding to the changed configuration when it commits the changes on the backup Routing Engine.

18.2R1	Starting in Junos OS Release 18.2R1, devices running Junos OS support configuring up to seven user-defined instances of the ephemeral configuration database. In earlier releases, you can configure up to eight user-defined instances.

RELATED DOCUMENTATION

Enable and Configure Instances of the Ephemeral Configuration Database

Example: Configure the Ephemeral Configuration Database Using NETCONF | 376

Unsupported Configuration Statements in the Ephemeral Configuration Database

IN THIS SECTION

The ephemeral database is an alternate configuration database. Juniper Extension Toolkit (JET) applications and NETCONF and Junos XML protocol client applications can simultaneously load and commit configuration changes to the ephemeral database with significantly greater throughput than when committing data to the candidate configuration database. To improve commit performance, the ephemeral commit process does not perform all of the operations and validations executed by the standard commit model. As a result, you cannot configure some features through the ephemeral database. For example, the ephemeral configuration database does not support configuring interface alias names.

Use Feature Explorer to confirm platform and release support for specific features.

Review the "Platform-Specific Ephemeral Database Behavior" on page 345 section for notes related to your platform.

The ephemeral configuration database does *not* support the following configuration statements. We've grouped the statements under their top-level configuration statement hierarchy. If a client attempts to configure an unsupported statement in an ephemeral instance, the server returns an error during the load operation.

[edit]

```
[edit apply-groups]
[edit access]
[edit chassis]
[edit dynamic-profiles]
[edit security] (SRX Series only)
```

[edit interfaces]

```
[edit interfaces interface-name unit logical-unit-number alias alias-name]
[edit interfaces interface-range]
```

[edit logical-systems]

```
[edit logical-systems logical-system-name interfaces interface-name unit logical-unit-number
alias alias-name]
[edit logical-systems logical-system-name policy-options prefix-list name apply-path path]
[edit logical-systems logical-system-name system processes routing]
```

NOTE: Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, you can configure MSTP, RSTP, and VSTP in the ephemeral configuration database on supported platforms.

[edit policy-options]

```
[edit policy-options prefix-list name apply-path path]
```

[edit protocols]

```
[edit protocols igmp]
[edit protocols mld]
```

NOTE: Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, you can configure MSTP, RSTP, and VSTP in the ephemeral configuration database on supported platforms.

[edit routing-instances]

NOTE: Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.4R1, you can configure MSTP, RSTP, and VSTP in the ephemeral configuration database on supported platforms.

[edit security]

```
[edit security group-vpn member ipsec vpn]
[edit security ssh-known-hosts host hostname]
```

[edit services]

```
[edit services ssl initiation profile]
[edit services ssl proxy profile]
[edit services ssl termination profile]
```

[edit system]

```
[edit system archival]
[edit system commit delta-export]
[edit system commit fast-synchronize]
[edit system commit notification]
[edit system commit peers]
[edit system commit peers-synchronize]
[edit system commit persist-groups-inheritance]
[edit system commit server]
[edit system compress-configuration-files]
[edit system configuration-database]
[edit system extensions]
[edit system fips]
[edit system host-name]
[edit system license]
[edit system login]
[edit system master-password]
[edit system max-configurations-on-flash]
[edit system radius-options]
[edit system regex-additive-logic]
[edit system scripts]
```

```
[edit system services extension-service notification allow-clients address]
[edit system time-zone]
```

Platform-Specific Ephemeral Database Behavior

Use Feature Explorer to confirm platform and release support for specific features.

Use the following table to review platform-specific behaviors for your platforms.

Table 17: Platform-Specific Behavior

Platform	Difference
SRX Series	• On SRX Series devices that support the ephemeral database, you cannot configure the `[edit security]` statement hierarchy in the ephemeral configuration database.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.4R1-EVO	Starting in Junos OS Evolved Release 23.4R1, Junos OS Evolved supports configuring MSTP, RSTP, and VSTP in the ephemeral configuration database on supported devices.
23.4R1	Starting in Junos OS Release 23.4R1, to configure MSTP, RSTP, or VSTP in the ephemeral configuration database, you must first configure the `ephemeral-db-support` statement at the `[edit protocols layer2-control]` hierarchy level in the static configuration database.
23.2R2	Starting in Junos OS Release 23.2R2, to configure MSTP, RSTP, or VSTP in the ephemeral configuration database, you must first configure the `ephemeral-db-support` statement at the `[edit protocols layer2-control]` hierarchy level in the static configuration database.
23.2R1	Starting in Junos OS Release 23.2R1, Junos OS supports configuring MSTP, RSTP, and VSTP in the ephemeral configuration database on supported devices.

RELATED DOCUMENTATION

Understanding the Ephemeral Configuration Database

Enable and Configure Instances of the Ephemeral Configuration Database

IN THIS SECTION

The ephemeral database is an alternate configuration database. It enables multiple client applications to concurrently load and commit configuration changes to a Junos device and with significantly greater throughput than when committing data to the candidate configuration database. Junos devices provide a default ephemeral database instance as well as the ability to enable and configure multiple user-defined instances of the ephemeral configuration database.

NETCONF and Junos XML protocol client applications and JET applications can update the ephemeral configuration database. The following sections detail how to enable instances of the ephemeral configuration database, configure the instances using NETCONF and Junos XML protocol operations, and display ephemeral configuration data in the CLI. The sections also discuss how to deactivate and then reactivate an ephemeral instance as well as delete an ephemeral instance. For information about using JET applications to configure the ephemeral configuration database, see the Juniper Extension Toolkit Documentation.

Enable Ephemeral Database Instances

The default ephemeral database instance is automatically enabled on Junos devices that support configuring the ephemeral database. However, you must configure any user-defined instances of the ephemeral configuration database before you can use the instance. See Feature Explorer to verify the hardware platforms and software releases that support the ephemeral database.

To enable a user-defined instance of the ephemeral configuration database:

1. Configure the name of the instance.

   The name must contain only alphanumeric characters, hyphens, and underscores, and it must not exceed 32 characters. You cannot use `default` as the name.

   ```
   [edit system configuration-database ephemeral]
   user@host# set instance instance-name
   ```

   NOTE: The priority of ephemeral database instances is determined by the order in which the configuration lists the instances. By default, newly configured instances are placed at the end of the list and have lower priority when resolving conflicting configuration statements. When you configure a new instance, you can specify its placement by using the `insert` command instead of the `set` command.

2. Commit the configuration.

   ```
   [edit system configuration-database ephemeral]
   user@host# commit
   ```

NOTE: When you commit statements at the `[edit system configuration-database ephemeral]` hierarchy level, all Junos processes must check and evaluate their complete configuration. As a result, there might be a spike in CPU utilization, potentially impacting other critical software processes.

Configure Ephemeral Database Options

You can configure several options for the ephemeral configuration database. You configure the options in the static configuration database.

1. (Optional) To disable the default instance of the ephemeral configuration database, configure the `ignore-ephemeral-default` statement.

   ```
   [edit system configuration-database ephemeral]
   user@host# set ignore-ephemeral-default
   ```

2. (Optional) Configure the commit synchronize model as asynchronous or synchronous.

The synchronous commit model is slower, but it is more reliable when synchronizing ephemeral configuration data to a backup Routing Engine or Virtual Chassis member.

```
[edit system configuration-database ephemeral]
user@host# set commit-synchronize-model (asynchronous | synchronous)
```

NOTE: Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the default commit synchronize model is `synchronous` and devices that enable GRES must use the synchronous model. In earlier releases, the default is `asynchronous`.

3. (Optional) When the device has graceful Routing Engine switchover (GRES) enabled and the ephemeral database uses the asynchronous commit synchronize model, configure the `allow-commit-synchronize-with-gres` statement to enable the device to synchronize an ephemeral instance to the other Routing Engine when you request a commit synchronize operation on that instance.

```
[edit system configuration-database ephemeral]
user@host# set allow-commit-synchronize-with-gres
```

NOTE: Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, we've deprecated the `allow-commit-synchronize-with-gres` statement and only the synchronous commit synchronize model supports synchronizing ephemeral data on devices that enable GRES.

4. Commit the configuration.

```
[edit system configuration-database ephemeral]
user@host# commit
```

NOTE: When you commit statements at the `[edit system configuration-database ephemeral]` hierarchy level, all Junos processes must check and evaluate their complete configuration. As a result, there might be a spike in CPU utilization, potentially impacting other critical software processes.

Enable MSTP, RSTP, and VSTP Configuration

On supported devices and releases, you can configure the following protocols in the ephemeral configuration database:

- Multiple Spanning Tree Protocol (MSTP)
- Rapid Spanning Tree Protocol (RSTP)
- VLAN Spanning Tree Protocol (VSTP)

Junos OS Evolved supports configuring these protocols in the ephemeral database in supported releases by default. However, on devices running Junos OS, you must configure the device to enable support for these protocols in the ephemeral database.

To enable users to configure MSTP, RSTP, and VSTP in the ephemeral database on devices running Junos OS:

1. In the static configuration database, configure the `ephemeral-db-support` statement at the `[edit protocols layer2-control]` hierarchy level.

   ```
   [edit protocols layer2-control]
   user@host# set ephemeral-db-support
   ```

2. Commit the configuration.

   ```
   [edit protocols layer2-control]
   user@host# commit
   ```

Open Ephemeral Database Instances

A client application must open an ephemeral database instance before viewing or modifying it. Within a NETCONF or Junos XML protocol session, a client application opens the ephemeral database instance by using the Junos XML protocol `<open-configuration>` operation with the appropriate child tags. Opening the ephemeral instance automatically acquires an exclusive lock on it.

- To open the default instance of the ephemeral database, a client application emits the `<open-configuration>` element and includes the `<ephemeral/>` child tag.

  ```
  <rpc>
      <open-configuration>
          <ephemeral/>
      </open-configuration>
  </rpc>
  ```

- To open a user-defined instance of the ephemeral database, a client application emits the `<open-configuration>` element and includes the `<ephemeral-instance>` element and the instance name.

```
<rpc>
    <open-configuration>
        <ephemeral-instance>instance-name</ephemeral-instance>
    </open-configuration>
</rpc>
```

Configure Ephemeral Database Instances

Client applications update the ephemeral configuration database using NETCONF and Junos XML protocol operations. Only a subset of the operations' attributes and options are available for use when updating the ephemeral configuration database. For example, options and attributes that reference groups, interface ranges, or commit scripts, or that roll back the configuration cannot be used with the ephemeral database.

Client applications load and commit configuration data to an open instance of the ephemeral configuration database. A client can load configuration data in any of the supported formats including Junos XML elements, formatted ASCII text, `set` commands, or JSON. By default, if a client disconnects from a session or closes the ephemeral database instance before committing new changes, the device discards any uncommitted data, but configuration data that has already been committed to the ephemeral database instance by that client is unaffected.

To update, commit, and close an open instance of the ephemeral configuration database, client applications perform the following tasks:

1. Load configuration data into the ephemeral database instance by performing one or more load operations.

   Client applications emit the `<load-configuration>` operation in a Junos XML protocol session or the `<load-configuration>` or `<edit-config>` operation in a NETCONF session and include the appropriate attributes and tags for the data.

```
<rpc>
    <load-configuration action="(merge | override | replace | set | update)" format="(text |
json | xml)">
        <!--configuration-data-->
    </load-configuration>
</rpc>
```

NOTE: The ephemeral configuration database supports the `update` attribute starting in Junos OS Release 21.1R1.

NOTE: The only acceptable format for `action="set"` is `"text"`. For more information about the `<load-configuration>` operation, see *<load-configuration>*.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <!--configuration-data-->
    </edit-config>
</rpc>
```

NOTE: The target value `<candidate/>` can refer to either the open configuration database, or if there is no open database, to the candidate configuration. If a client application issues the Junos XML protocol `<open-configuration>` operation to open an ephemeral instance before executing the `<edit-config>` operation, the device performs the `<edit-config>` operation on the open instance of the ephemeral configuration database. Otherwise, the device performs the operation on the candidate configuration.

2. (Optional) Review the updated configuration in the open ephemeral instance by emitting the `<get-configuration/>` operation in a Junos XML protocol session or the `<get-configuration/>` or `<get-config>` operation in a NETCONF session.

```
<rpc>
    <get-configuration format="(json | set | text | xml)"/>
</rpc>
```

```
<rpc>
    <get-config>
        <source>
            <candidate/>
        </source>
    </get-config>
</rpc>
```

3. Commit the configuration changes by emitting the `<commit-configuration/>` operation in a Junos XML protocol session or the `<commit-configuration/>` or `<commit/>` operation in a NETCONF session.

   Include the `<synchronize/>` tag in the `<commit-configuration>` element to synchronize the data to a backup Routing Engine or to other members of a Virtual Chassis.

```
<rpc>
    <commit-configuration/>
</rpc>
```

```
<rpc>
    <commit-configuration>
        <synchronize/>
    </commit-configuration>
</rpc>
```

```
<rpc>
    <commit/>
</rpc>
```

NOTE: Starting in Junos OS Release 22.1R1, you can automatically synchronize an ephemeral instance's configuration to the other Routing Engine every time you commit the instance. To automatically synchronize an instance, include the `synchronize` statement at the `[edit system commit]` hierarchy level within that ephemeral instance's configuration.

NOTE: After a client application commits changes to the ephemeral database instance, the device merges the ephemeral data into the active configuration according to the rules of prioritization.

4. Repeat steps 1 through 3 for any subsequent updates to the ephemeral database instance.

5. Close the ephemeral database instance, which releases the exclusive lock.

```
<rpc>
    <close-configuration/>
</rpc>
```

Display Ephemeral Configuration Data in the CLI

The active device configuration is a merged view of the static and ephemeral configuration databases. However, when you display the configuration using the `show configuration` command in operational mode, the output does not include ephemeral configuration data. To display the data in a specific ephemeral database instance or display a merged view of the static and ephemeral configuration databases, use variations of the `show ephemeral-configuration` CLI command.

Table 18 on page 353 summarizes the `show ephemeral-configuration` commands.

Table 18: `show ephemeral-configuration` Command

Action	`show ephemeral-configuration` Command
View the configuration data in the default ephemeral instance.	**show ephemeral-configuration instance default**
View the configuration data in a user-defined ephemeral instance.	**show ephemeral-configuration instance *instance-name***
View the complete post-inheritance configuration merged with the configuration data in all instances of the ephemeral database.	**show ephemeral-configuration merge**
Specify the scope of the configuration data to display in a specific ephemeral instance. Append the statement path of the requested hierarchy to the command.	**show ephemeral-configuration instance *instance-name* *hierarchy-to-view*** For example: **show ephemeral-configuration instance default protocols mpls**

Deactivate Ephemeral Database Instances

When you enable and configure an ephemeral instance, the Junos device stores the instance's configuration data in files, which is similar to the operation of the static configuration database. You can deactivate a specific ephemeral instance within the static configuration database. When you deactivate an instance and commit the configuration, the device preserves the instance's configuration data and files, but it does not merge the instance's configuration with the static configuration database. If you later reactivate the instance in the static configuration database, the device merges the instance's existing configuration data with the static configuration database.

NOTE: On devices running Junos OS Release 22.1R1 or later and devices running Junos OS Evolved, when you deactivate the entire `[edit system configuration-database ephemeral]` hierarchy level and commit the configuration, the device deletes the files and corresponding configuration data for all user-defined ephemeral instances. In earlier Junos OS releases, the device preserves the files and configuration data; however, the device does not merge the configuration data with the static configuration database. Deactivating the hierarchy does not affect the default ephemeral instance's files.

To deactivate the default ephemeral instance or a user-defined ephemeral instance in the static configuration database:

1. Deactivate the ephemeral database instance.

   - Deactivate the default ephemeral instance by configuring the `ignore-ephemeral-default` statement.

     ```
     [edit system configuration-database ephemeral]
     user@host# set ignore-ephemeral-default
     ```

   - Deactivate a user-defined ephemeral instance by issuing the `deactivate` command and specifying the instance name.

     ```
     [edit system configuration-database ephemeral]
     user@host# deactivate instance instance-name
     ```

2. Commit the configuration.

   ```
   [edit system configuration-database ephemeral]
   user@host# commit
   ```

To reactivate an ephemeral instance and thus merge its configuration with the static configuration database again:

1. Activate the ephemeral database instance.

   - Activate the default ephemeral instance by deleting the `ignore-ephemeral-default` statement.

     ```
     [edit system configuration-database ephemeral]
     user@host# delete ignore-ephemeral-default
     ```

   - Activate a user-defined ephemeral instance by issuing the `activate` command and specifying the instance name.

     ```
     [edit system configuration-database ephemeral]
     user@host# activate instance instance-name
     ```

2. Commit the configuration.

   ```
   [edit system configuration-database ephemeral]
   user@host# commit
   ```

Delete Ephemeral Database Instances

When you enable and configure an ephemeral instance, the Junos device stores the instance's configuration data in files, which is similar to the operation of the static configuration database. On devices running Junos OS Release 22.1R1 or later and devices running Junos OS Evolved, when you delete an ephemeral instance from the static configuration database and commit the configuration, the device also deletes the ephemeral instance's files and corresponding configuration data. Thus, if you later configure an ephemeral instance with the same name, there is no existing configuration data associated with this instance name.

However, in earlier Junos OS releases, when you delete an ephemeral instance, the device preserves the ephemeral instance's files. Thus, if you later configure an ephemeral instance with the same name, the device restores the configuration data associated with the instance name from the corresponding files. If you delete an ephemeral instance in an earlier release, we recommend that you delete the ephemeral instance's configuration data before you delete the instance from the static configuration database.

To delete the default ephemeral instance or a user-defined ephemeral instance from the static configuration database:

1. Delete the ephemeral database instance.

- Delete the default ephemeral instance by configuring the `delete-ephemeral-default` and `ignore-ephemeral-default` statements.

  ```
  [edit system configuration-database ephemeral]
  user@host# set delete-ephemeral-default
  user@host# set ignore-ephemeral-default
  ```

  NOTE: Devices running Junos OS Release 22.1R1 or later and devices running Junos OS Evolved support the `delete-ephemeral-default` statement.

- Delete a user-defined ephemeral instance by issuing the `delete` command and specifying the instance name.

  ```
  [edit system configuration-database ephemeral]
  user@host# delete instance instance-name
  ```

2. Commit the configuration.

   ```
   [edit system configuration-database ephemeral]
   user@host# commit
   ```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the default commit synchronize model is `synchronous`. In earlier releases, the default is `asynchronous`.
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES must use the synchronous commit synchronize model.
22.1R1	Starting in Junos OS Release 22.1R1, when you deactivate the entire `[edit system configuration-database ephemeral]` hierarchy level, Junos OS deletes the files and corresponding configuration data for all user-defined ephemeral instances. In earlier releases, the files and configuration data are preserved; however, the configuration data is not merged with the static configuration database.

22.1R1	Starting in Junos OS Release 22.1R1, when you delete an ephemeral instance in the static configuration database, the instance's configuration files are also deleted. In earlier releases, the configuration files are preserved.
18.2R1	Starting in Junos OS Release 18.2R1, the `show ephemeral-configuration` operational mode command uses a different syntax and supports filtering for displaying specific hierarchy levels.
18.1R1	Starting in Junos OS Release 18.1R1, the ephemeral configuration database supports loading configuration data using the `<load-configuration>` `action` attribute values of `override` and `replace` in addition to the previously supported values of `merge` and `set`.

RELATED DOCUMENTATION

Example: Configure the Ephemeral Configuration Database Using NETCONF | **376**

Understanding the Ephemeral Configuration Database

Commit and Synchronize Ephemeral Configuration Data Using the NETCONF or Junos XML Protocol

ephemeral

show ephemeral-configuration

Commit and Synchronize Ephemeral Configuration Data Using the NETCONF or Junos XML Protocol

IN THIS SECTION

Commit an Ephemeral Instance Overview

The ephemeral database is an alternate configuration database. It enables NETCONF and Junos XML protocol client applications to simultaneously load and commit configuration changes on Junos devices and with significantly greater throughput than when committing data to the candidate configuration database. Client applications can commit the configuration data in an open instance of the ephemeral configuration database so that it becomes part of the active configuration on the device. When you commit ephemeral configuration data on a device, the device's active configuration is a merged view of the static and ephemeral configuration databases.

CAUTION: The ephemeral commit model validates the syntax but not the semantics, or constraints, of the configuration data committed to the ephemeral database. You must validate all configuration data before loading it into the ephemeral database and committing it on the device. Committing invalid configuration data can cause Junos processes to restart or stop responding and result in disruption to the system or network.

After a client application commits an ephemeral instance, the device merges the configuration data into the ephemeral database. The system processes parse the configuration and then merge the ephemeral data with the data in the active configuration. If there are conflicting statements in the static and ephemeral configuration databases, the device merges the data according to specific rules of prioritization. The database priority, from highest to lowest, is as follows:

1. Statements in a user-defined instance of the ephemeral configuration database.

   If the device uses multiple user-defined ephemeral instances, it determines the priority by the order in which the instances are configured at the `[edit system configuration-database ephemeral]` hierarchy level, running from highest to lowest priority.

2. Statements in the default ephemeral database instance.

3. Statements in the static configuration database.

NOTE: Applications can simultaneously load and commit data to different ephemeral database instances in addition to the static configuration database. However, the device processes the commits sequentially. As a result, the commit to a specific database might be delayed, depending on the processing order.

NOTE: If you commit ephemeral configuration data that is invalid or results in undesirable network disruption, you must remove the problematic data from the database. You can delete the data, or if necessary, you can reboot the device, which deletes the configuration data in all instances of the ephemeral configuration database.

The active device configuration is a merged view of the static and ephemeral configuration databases. However, when you display the configuration using the `show configuration` command in operational mode, the output does not include ephemeral configuration data. To display the data in a specific ephemeral database instance or display a merged view of the static and ephemeral configuration databases, use variations of the `show ephemeral-configuration` CLI command.

How to Commit an Ephemeral Instance

Client applications can commit the configuration data in an open instance of the ephemeral configuration database so that it becomes part of the active configuration on the device. To commit the configuration, use the `<commit-configuration/>` operation in a Junos XML protocol session or the `<commit-configuration/>` or `<commit/>` operation in a NETCONF session.

In a Junos XML protocol session, a client application commits the configuration data in an open instance of the ephemeral configuration database by performing a `<commit-configuration/>` operation (just as for the candidate configuration).

```
<rpc>
    <commit-configuration/>
</rpc>
```

The Junos XML protocol server reports the results of the commit operation in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the commit operation succeeds, the `<routing-engine>` element encloses the `<commit-success/>` tag and the `<name>` element, which specifies the target Routing Engine.

```
<rpc-reply xmlns:junos="URL">
   <commit-results>
       <routing-engine>
           <name>routing-engine-name</name>
           <commit-success/>
       </routing-engine>
   </commit-results>
</rpc-reply>
```

In a NETCONF session, a client application commits the configuration data in an open instance of the ephemeral configuration database by performing a `<commit/>` or `<commit-configuration/>` operation (just as for the candidate configuration).

```
<rpc>
    <commit/>
</rpc>
]]> ]]>
```

```
<rpc>
    <commit-configuration/>
</rpc>
]]> ]]>
```

The NETCONF server confirms that the commit operation was successful by returning the `<ok/>` tag in an `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <ok/>
</rpc-reply>
]]> ]]>
```

If the commit operation fails, the NETCONF server returns the `<rpc-reply>` element and `<rpc-error>` child element, which explains the reason for the failure.

The only variant of the commit operation supported for the ephemeral database is synchronizing the configuration, as described in "Overview of Synchronizing an Ephemeral Instance" on page 360.

Overview of Synchronizing an Ephemeral Instance

Dual Routing Engine devices and Virtual Chassis systems do not automatically synchronize ephemeral configuration data when you commit an ephemeral instance. You can synchronize the data in an ephemeral instance on a per-commit or per-session basis. You can also configure an ephemeral instance to synchronize its data every time you commit the instance. The environment determines where the data is synchronized, for example:

- A dual Routing Engine device synchronizes the ephemeral instance to the backup Routing Engine.
- An MX Series Virtual Chassis synchronizes the ephemeral instance only to the backup device's primary Routing Engine.
- An EX Series Virtual Chassis synchronizes the ephemeral instance to all members switches.

NOTE: Virtual Chassis environments do not support synchronizing the ephemeral configuration database to the backup Routing Engine on the respective Virtual Chassis member.

See the following sections for instructions on synchronizing ephemeral instances:

- "How to Configure GRES-Enabled Devices to Synchronize Ephemeral Configuration Data" on page 363
- "How to Synchronize an Ephemeral Instance on a Per-Commit Basis" on page 364
- "How to Synchronize an Ephemeral Instance on a Per-Session Basis" on page 365
- "How to Automatically Synchronize an Ephemeral Instance upon Commit" on page 366
- "How to Configure Failover Configuration Synchronization for the Ephemeral Database" on page 367

The ephemeral database supports two commit synchronize models: asynchronous and synchronous. Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the ephemeral database uses the synchronous model by default. In earlier releases, the asynchronous model is the default. You can configure the `commit-synchronize-model` statement to explicitly configure the model.

```
[edit system configuration-database ephemeral]
user@host# set commit-synchronize-model (asynchronous | synchronous)
```

In the asynchronous commit model, the NETCONF or Junos XML protocol server first commits the configuration on the local Routing Engine and then notifies the other Routing Engine or Virtual Chassis device. The requesting Routing Engine does not wait for the other Routing Engine or Virtual Chassis member to first synchronize and commit the configuration.

Synchronous commit operations are slower but more reliable than asynchronous commit operations. We recommend that you use the synchronous commit model on devices that have graceful Routing Engine switchover (GRES) or nonstop active routing (NSR) enabled. In the synchronous model, the primary Routing Engine or MX Virtual Chassis primary device generally only completes its commit operation if the commit on the backup Routing Engine or Virtual Chassis backup device is successful.

When you synchronize an ephemeral instance, the Junos XML protocol server reports the results of the commit operation for the local Routing Engine in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the commit operation succeeds, the `<routing-engine>` element encloses the `<commit-success/>` tag and the `<name>` element, which specifies the target Routing Engine.

The server reply includes additional tags that depend on the commit synchronize model used by the database.

- If the ephemeral database uses the synchronous model, the server reply includes a second `<routing-engine>` element for the commit operation on the other Routing Engine.
- If the ephemeral database uses the asynchronous model, the server includes the `<commit-synchronize-server-success>` element. This tag indicates that the synchronize operation is scheduled on the other Routing Engine or Virtual Chassis members and provides the estimated time in seconds required for the operation to complete.

For example:

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
  <commit-synchronize-server-success>
    <current-job-id>0</current-job-id>
    <number-of-jobs>1</number-of-jobs>
    <estimated-time>60</estimated-time>
  </commit-synchronize-server-success>
</rpc-reply>
```

For synchronous commit operations, the RPC reply indicates the success or failure of the commit operation on the other Routing Engine or Virtual Chassis members. For asynchronous commit operations, the device records the success or failure of the scheduled commit operations in the system log file. You must configure the device to log events of the given facility and severity level corresponding to these operations. See the System Log Explorer for the various ephemeral database events and the facility and severity levels required to log them.

Similarly, in NETCONF sessions, the server confirms that the commit operation was successful by returning the `<ok/>` tag in an `<rpc-reply>` tag element. Depending on the device configuration, the response might also include the `<commit-results>` element for synchronous commit synchronize operations or the `<commit-synchronize-server-success>` element for asynchronous commit synchronize operations. For example:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
  <commit-synchronize-server-success>
    <current-job-id>0</current-job-id>
    <number-of-jobs>1</number-of-jobs>
    <estimated-time>60</estimated-time>
  </commit-synchronize-server-success>
</rpc-reply>
]]>]]>
```

NOTE: The device does not synchronize the ephemeral configuration database to the other Routing Engine or Virtual Chassis members when you issue the `commit synchronize` command on the static configuration database.

How to Configure GRES-Enabled Devices to Synchronize Ephemeral Configuration Data

The ephemeral database supports two commit synchronize models: asynchronous and synchronous. To ensure a GRES-enabled device synchronizes ephemeral configuration data when you request a commit synchronize operation on an ephemeral instance, you must use one of the following methods:

- Use the synchronous model
- Use the asynchronous model and configure the `allow-commit-synchronize-with-gres` statement

We recommend that you use the synchronous model on devices that enable GRES. Additionally, starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the ephemeral database uses the synchronous model by default and devices that enable GRES must use the synchronous model. In earlier releases, the asynchronous model supports synchronizing ephemeral data on GRES-enabled devices provided that you configure the `allow-commit-synchronize-with-gres` statement. However, we do not recommend using the asynchronous model on devices that enable GRES.

To configure GRES-enabled devices to synchronize ephemeral configuration data:

1. Configure the commit model that the ephemeral database uses to perform commit synchronize operations.

   - (Recommended) To use the synchronous commit model, configure the `synchronous` option.

     ```
     [edit system configuration-database ephemeral]
     user@host# set commit-synchronize-model synchronous
     ```

- Alternatively, to use the asynchronous commit model in Junos OS Release 25.2 and earlier or Junos OS Evolved Release 25.2 and earlier, configure the `asynchronous` option and the `allow-commit-synchronize-with-gres` statement.

  ```
  [edit system configuration-database ephemeral]
  user@host# set commit-synchronize-model asynchronous
  user@host# set allow-commit-synchronize-with-gres
  ```

2. Commit the configuration.

   ```
   [edit]
   user@host# commit synchronize
   ```

How to Synchronize an Ephemeral Instance on a Per-Commit Basis

You can synchronize an ephemeral instance across Routing Engines or Virtual Chassis members for a given commit operation on that instance.

To synchronize an ephemeral instance on a per-commit basis:

1. Open the ephemeral instance.

   ```
   <rpc>
      <open-configuration>
         <ephemeral-instance>instance-name</ephemeral-instance>
      </open-configuration>
   </rpc>
   ```

2. Configure the ephemeral instance.

   ```
   <rpc>
       <load-configuration>
           <!--configuration-data-->
       </load-configuration>
   </rpc>
   ```

3. Commit and synchronize the instance by enclosing the empty `<synchronize/>` tag in the `<commit-configuration>` and `<rpc>` tag elements.

```
<rpc>
    <commit-configuration>
        <synchronize/>
    </commit-configuration>
</rpc>
```

4. Repeat steps 2 and 3, as appropriate.
5. Close the ephemeral instance.

```
<rpc>
   <close-configuration/>
</rpc>
```

How to Synchronize an Ephemeral Instance on a Per-Session Basis

You can synchronize an ephemeral instance across Routing Engines or Virtual Chassis members for all commit operations performed for the duration that the ephemeral instance is open, which we are loosely referring to as a session. This session should not be confused with the NETCONF or Junos XML protocol session. Synchronizing the instance on a per-session basis enables you to execute multiple load and commit operations and ensure that each commit operation automatically synchronizes the instance until you close it.

To synchronize an ephemeral instance for all commit operations performed for the duration that the instance is open:

1. Open the ephemeral instance, and include the `<commit-synchronize/>` tag.

```
<rpc>
    <open-configuration>
        <ephemeral-instance>instance-name</ephemeral-instance>
        <commit-synchronize/>
    </open-configuration>
</rpc>
```

2. Configure the ephemeral instance.

```
<rpc>
    <load-configuration>
```

```
        <!--configuration-data-->
    </load-configuration>
</rpc>
```

3. Commit the instance, which also synchronizes it to the other Routing Engine or Virtual Chassis members.

```
<rpc>
    <commit-configuration/>
</rpc>
```

4. Repeat steps 2 and 3, as appropriate.
5. Close the ephemeral instance.

```
<rpc>
    <close-configuration/>
</rpc>
```

How to Automatically Synchronize an Ephemeral Instance upon Commit

On devices running Junos OS Release 22.1R1 or later and devices running Junos OS Evolved, you can configure an ephemeral instance so that it synchronizes its configuration across Routing Engines or Virtual Chassis members every time you commit the instance.

To configure the ephemeral instance to synchronize every time you commit the instance:

1. Open the ephemeral instance.

```
<rpc>
   <open-configuration>
      <ephemeral-instance>instance-name</ephemeral-instance>
   </open-configuration>
</rpc>
```

2. Configure the ephemeral instance to include the `synchronize` statement at the `[edit system commit]` hierarchy level.

```
<rpc>
    <load-configuration>
        <configuration>
            <system>
```

```
                <commit>
                    <synchronize/>
                </commit>
            </system>
        </configuration>
    </load-configuration>
</rpc>
```

3. Commit the instance, which also synchronizes its configuration to the other Routing Engine.

```
<rpc>
    <commit-configuration/>
</rpc>
```

4. Close the ephemeral instance.

```
<rpc>
    <close-configuration/>
</rpc>
```

After you add the `synchronize` statement at the `[edit system commit]` hierarchy level in the ephemeral instance's configuration, the device automatically synchronizes the instance to the other Routing Engine or Virtual Chassis members whenever you commit that instance, provided that the device meets the necessary requirements for synchronizing the database.

How to Configure Failover Configuration Synchronization for the Ephemeral Database

Dual Routing Engine devices and MX Series Virtual Chassis support failover configuration synchronization for the ephemeral database. Failover configuration synchronization helps ensure that the configuration database is synchronized between Routing Engines in the event of a Routing Engine switchover. To enable failover synchronization, you configure the `commit synchronize` statement at the `[edit system]` hierarchy level in the static configuration database.

When you configure the `commit synchronize` statement in the static configuration database, it has the following effects:

- The device synchronizes its static configuration database to the backup Routing Engine or MX Virtual Chassis backup device during a commit operation.

  NOTE: If you configure the `commit synchronize` statement in the static configuration database, the device does not automatically synchronize an ephemeral instance to the backup device when you commit the static configuration database or when you commit the instance.

- Starting in Junos OS Release 20.2R1, a backup Routing Engine or an MX Virtual Chassis backup device synchronizes both its static and ephemeral configuration databases when it synchronizes with the primary device. In earlier releases, a backup Routing Engine only synchronizes the static configuration database.

NOTE: For failover synchronization, the backup Routing Engine and the MX Virtual Chassis backup device only synchronize the ephemeral configuration database with the primary device if both the backup device and the primary device are running the same software version.

When you configure the `commit synchronize` statement on the primary and backup Routing Engines, the backup Routing Engine synchronizes its configuration with the primary Routing Engine in the following scenarios:

- You remove and reinsert the backup Routing Engine
- You reboot the backup Routing Engine
- The device performs a graceful Routing Engine switchover
- There is a manual change in roles
- You insert a new backup Routing Engine that has the `commit synchronize` statement configured

On a dual Routing Engine system, the backup Routing Engine synchronizes its configuration databases with the primary Routing Engine. In an MX Series Virtual Chassis, the primary Routing Engine on the backup device synchronizes its configuration databases with the primary Routing Engine on the primary device.

To enable failover configuration synchronization for both the static and ephemeral databases on supported devices running Junos OS Release 20.2R1 or later or devices running Junos OS Evolved:

1. Configure the `synchronize` statement in the static configuration database.

```
[edit]
user@host# set system commit synchronize
```

2. Commit the configuration.

```
[edit]
user@host# commit synchronize
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, the default commit synchronize model is `synchronous`. In earlier releases, the default is `asynchronous`.
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, devices that enable GRES or NSR must use the synchronous commit synchronize model.
20.2R1	Starting in Junos OS Release 20.2R1, when you configure the `synchronize` statement at the `[edit system commit]` hierarchy level in the static configuration database, the backup Routing Engine synchronizes both the static and ephemeral configuration databases when it synchronizes with the primary Routing Engine. In earlier releases, the backup Routing Engine only synchronizes the static configuration database.

RELATED DOCUMENTATION

Enabling and Configuring Instances of the Ephemeral Configuration Database

Understanding the Ephemeral Configuration Database

Managing Ephemeral Configuration Database Space

SUMMARY

Configure options for ephemeral database instances to more effectively manage the amount of space that the database uses.

IN THIS SECTION

Junos devices maintain versions of ephemeral configuration database objects with every commit. Thus, any change to the ephemeral database, whether it is an addition, modification, or deletion, increases the size of the database. As a result, the database only increases in size over time. Depending on the size of the ephemeral configuration and the changes to the database, the database can consume a lot of disk space, become fragmented, and could potentially run into the maximum database size. You can manage the space that an ephemeral database instance uses by configuring different options.

In supported releases, Junos devices, by default, perform cyclic versioning when you commit an ephemeral instance. Cyclic versioning reclaims the space occupied by objects deleted in a previous database version. To manage the space consumed by the ephemeral database, you can configure the device to:

- Adjust cyclic versioning as appropriate for your operations.
- Resize an ephemeral database when it meets specific criteria.

Benefits of Cyclic Versioning and Resizing

- More efficiently manage ephemeral configuration database space as required for a given environment.
- Reduce database fragmentation for improved performance.
- Prevent an ephemeral configuration database from running into the maximum database size.

Understanding Cyclic Versioning

Junos devices maintain versioning for ephemeral database objects, and as a result, the database also retains and stores deleted objects. A deletion is characterized by:

- Explicitly deleting the configuration.
- Changing the value of a configuration attribute.
- Reordering elements during a load update operation.

Cyclic versioning reclaims the space occupied by objects that were deleted in a previous version of the database. The cyclic version value determines the ephemeral database version in which the system reclaims deleted objects during a commit operation. The default cyclic version value for each ephemeral database instance is 10. Thus, on devices that support cyclic versioning, the system, by default, reclaims the space occupied by deleted configuration objects with each commit. You can modify the setting on a per-instance basis. To disable cyclic versioning, set the cyclic version value to 0.

For example, if you use the default cyclic version value of 10, then:

- After the 11th commit (version 11), the device reclaims the space occupied by objects that were deleted in version 1.
- After the 12th commit (version 12), the device reclaims the space occupied by objects that were deleted in version 2.
- After the 13th commit (version 13), the device reclaims the space occupied by objects that were deleted in version 3.

This process continues with each subsequent commit operation. As illustrated in the previous example, the version from which the system reclaims deleted objects during the current commit operation is:

```
version to reclaim = current version - cyclic version
```

NOTE: When the system resizes the database, the system keeps only the active configuration objects and resets the version for each object to the latest version. As a result, the system does not reclaim deleted objects again until after you execute commit operations equal to the cyclic version value.

In earlier releases and on devices that do not use cyclic versioning, the ephemeral database default behavior is to purge the database when it reaches the maximum allowable version. A purge operation reclaims the space used by deleted objects but requires all processes to read the full configuration. A database purge operation involves:

- Creating a new database.
- Copying only the active configuration objects from the current database into the new database.
- Setting the version for all active configuration objects in the new database to version 1.

Understanding Ephemeral Database Resizing

Resizing an ephemeral database might be necessary if cyclic versioning is enabled and you make frequent changes to the database that involve deleting or reordering elements. On devices that support cyclic versioning, the system automatically reclaims the space occupied by deleted objects during a commit operation. However, the system might or might not reallocate the freed space for new configuration objects when you update the database. If the system does not reallocate the space, then the database can become fragmented over time. Resizing an ephemeral database reclaims the space occupied by all deleted objects and defragments the database, which can improve performance.

A database resize operation involves:

- Creating a new database.
- Copying only the active configuration objects from the current database into the new database.
- Setting the version for all active configuration objects in the new database to the latest version.

As with the static configuration database, you can configure Junos devices to resize the ephemeral configuration database. After you configure database resizing, Junos devices resize the ephemeral database during a commit operation if the database's space exceeds the specified thresholds. You can modify the thresholds for each ephemeral instance.

The system resizes the database when the database size meets the criteria for both of the following configuration statements:

- `database-size-diff`—Minimum difference between the database size and the actual usage. Default is 100 MB.
- `database-size-on-disk`—Minimum configuration database size on disk. Default is 450 MB.

For example, suppose you configure the device to use the default values. Then the system resizes the database when the database size on disk exceeds 450 MB *and* the database size is 100 MB greater than the actual database usage.

For information about configuring database resizing, see "Resize an Ephemeral Database Instance" on page 374.

Use the `show system configuration database usage` command to display the database's disk space usage. The command displays the current database size on disk, the actual database usage, and the maximum size of the database.

```
user@host> show system configuration database usage ephemeral-instance default
Maximum size of the database: 692.49 MB
Current database size on disk: 1.50 MB
Actual database usage: 1.49 MB
Available database space: 691.01 MB
```

Configure Cyclic Versioning

Junos devices, by default, use a cyclic version value of 10. When configuring the cyclic version value, the best practice is to use a smaller value if you perform frequent commit operations for scaled configurations that reorder elements or delete many objects or attributes. A smaller value causes the device to store deleted objects for fewer versions of the database and thus use less disk space overall. In such cases, we recommend a value of 2 or 3. Otherwise, you can use a larger cyclic version value, such as the default value of 10.

NOTE: If a Junos process misses reading more commits than the configured cyclic version value, it must read the full configuration because the delta between the versions is no longer available. This effect might happen more frequently if you configure smaller cyclic version values.

To specify the cyclic version value that the device uses to reclaim the space occupied by deleted objects during a commit operation:

1. Configure the cyclic version value for the default ephemeral instance.

   ```
   [edit system configuration-database ephemeral]
   user@host# set cyclic-version-for-ephemeral-default version
   ```

   For example:

   ```
   [edit system configuration-database ephemeral]
   user@host# set cyclic-version-for-ephemeral-default 8
   ```

2. Configure the cyclic version value for a user-defined ephemeral instance.

   ```
   [edit system configuration-database ephemeral]
   user@host# set instance instance-name cyclic-version verison
   ```

   For example:

   ```
   [edit system configuration-database ephemeral]
   user@host# set instance eph1 cyclic-version 3
   ```

3. Commit the configuration.

```
[edit system configuration-database ephemeral]
user@host# commit
```

Resize an Ephemeral Database Instance

Junos devices do not automatically resize an ephemeral database. You can configure the device to resize an ephemeral database during a commit operation when the database size meets certain thresholds. You can enable resizing and use either the default values or custom values that are appropriate for your environment. To configure resizing:

1. Enable resizing for the default ephemeral instance.

    - To use the default values, configure the top-level `resize-ephemeral-default` statement.

    ```
    [edit system configuration-database ephemeral]
    user@host# set resize-ephemeral-default
    ```

    - To use custom values, configure the database size difference and the database size on disk in MB.

    ```
    [edit system configuration-database ephemeral]
    user@host# set resize-ephemeral-default database-size-diff size
    user@host# set resize-ephemeral-default database-size-on-disk size
    ```

    For example:

    ```
    [edit system configuration-database ephemeral]
    user@host# set resize-ephemeral-default database-size-diff 50
    user@host# set resize-ephemeral-default database-size-on-disk 600
    ```

2. Enable resizing for a user-defined instance.

    - To use the default values, configure the top-level `resize` statement.

    ```
    [edit system configuration-database ephemeral]
    user@host# set instance instance-name resize
    ```

- To use custom values, configure the database size difference and the database size on disk in MB.

  ```
  [edit system configuration-database ephemeral]
  user@host# set instance instance-name resize database-size-diff size
  user@host# set instance instance-name resize database-size-on-disk size
  ```

  For example:

  ```
  [edit system configuration-database ephemeral]
  user@host# set instance eph1 resize database-size-diff 150
  user@host# set instance eph1 resize database-size-on-disk 500
  ```

3. Commit the configuration.

   ```
   [edit system configuration-database ephemeral]
   user@host# commit
   ```

After you configure the device to resize the database, the device resizes the database after a commit operation on that database when it meets the specified criteria. After successfully resizing the database, the device emits the following message:

```
Database resize completed
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.2R1 and 23.2R1-EVO	Starting in Junos OS Release 23.2R1 and Junos OS Evolved Release 23.2R1, Junos devices automatically perform cyclic versioning for the ephemeral configuration database. In earlier releases, the device purges deleted objects from the database only when it reaches the maximum version number.

RELATED DOCUMENTATION

ephemeral

Example: Configure the Ephemeral Configuration Database Using NETCONF

IN THIS SECTION

The ephemeral database is an alternate configuration database that enables client applications to simultaneously load and commit configuration changes on Junos devices and with significantly greater throughput than when committing data to the candidate configuration database. This example shows how to enable an instance of the ephemeral configuration database and make updates to that instance in a NETCONF session.

Requirements

This example uses the following software components:

- A device running Junos OS or a device running Junos OS Evolved that supports configuring the ephemeral database.

Before you begin:

- Enable the NETCONF-over-SSH service on the Junos device.

Overview

Multiple NETCONF and Junos XML protocol client applications can simultaneously load and commit configuration changes to a Junos device by using ephemeral database instances. This example enables the ephemeral database instance `eph1` and then configures the instance through a NETCONF session.

A client application must open an instance of the ephemeral configuration database in order to view or modify it. After establishing a NETCONF session, the client opens the ephemeral instance by using the Junos XML protocol `<open-configuration>` operation. The element encloses the `<ephemeral-instance>` child tag and the name of the instance. Opening the ephemeral instance automatically acquires an exclusive lock on it.

The client then loads configuration data in text format into the `eph1` ephemeral instance. Because the configuration data is in text format, the `<load-configuration>` operation must include the `format="text"` attribute, and a `<configuration-text>` element must enclose the configuration data.

The client application commits the configuration changes in the ephemeral instance by emitting the Junos XML protocol `<commit-configuration>` operation. The `<load-configuration> action="merge"` attribute only determines how the the device merges configuration data into that instance of the ephemeral database. After you commit the changes to the ephemeral instance, the device merges the configuration data into the active configuration according to the rules of prioritization. If the different configuration databases have conflicting data, statements in the `eph1` instance have a higher priority than statements in the default ephemeral instance or the static configuration database. If other user-defined ephemeral instances are in use, the priority is determined by the order in which the instances are listed in the configuration at the `[edit system configuration-database ephemeral]` hierarchy level.

The `<close-configuration/>` operation closes the open ephemeral instance and releases the exclusive lock. The device retains the committed ephemeral data until the device is rebooted or the data is deleted. If you reboot the device, the system deletes the configuration data in the `eph1` ephemeral instance as well as the data in all other ephemeral instances.

Configuration

IN THIS SECTION

Enable the Ephemeral Database Instance

Step-by-Step Procedure

To enable the ephemeral database instance:

1. Configure the name of the instance.

```
[edit]
user@host# set system configuration-database ephemeral instance eph1
```

2. Commit the configuration.

```
[edit]
user@host# commit
```

Results

From configuration mode, confirm your configuration by entering the `show system configuration-database` command. If the output does not display the intended configuration, repeat the instructions in this example to correct the configuration.

```
[edit]
user@host# show system configuration-database
ephemeral {
    instance eph1;
}
```

Configure the Ephemeral Database Instance

Step-by-Step Procedure

To configure the ephemeral database instance and commit the changes from within a NETCONF session, a client application performs the following steps:

1. Opens the ephemeral database instance.

```
<rpc>
    <open-configuration>
        <ephemeral-instance>eph1</ephemeral-instance>
    </open-configuration>
</rpc>
]]>]]>
```

2. Loads the configuration data into the open ephemeral instance, and includes the appropriate tags and attributes for that data.

```
<rpc>
    <load-configuration action="merge" format="text">
```

```
          <configuration-text>
              protocols {
                  mpls {
                      label-switched-path to-hastings {
                          to 192.0.2.1;
                      }
                  }
              }
          </configuration-text>
      </load-configuration>
  </rpc>
  ]]>]]>
```

The NETCONF server indicates a successful `<load-configuration>` operation by returning an empty `<ok/>` tag enclosed within the `<load-configuration-results>` and `<rpc-reply>` elements.

3. Commits the configuration, provided the `<load-configuration>` operation does not generate any errors.

```
<rpc>
    <commit-configuration/>
</rpc>
]]>]]>
```

4. Closes the ephemeral database instance.

```
<rpc>
    <close-configuration/>
</rpc>
]]>]]>
```

Results

If there are no errors when opening or closing the database, the NETCONF server returns an empty `<rpc-reply>` element in response to the requests. The NETCONF server indicates a successful `<load-configuration>` operation by returning an empty `<ok/>` tag enclosed within the `<load-configuration-results>` and `<rpc-reply>` elements. Similarly, the NETCONF server indicates a successful `<commit-configuration>` operation by returning an empty `<ok/>` tag enclosed in an `<rpc-reply>` element.

Verification

IN THIS SECTION

Verify the Commit

Purpose

The NETCONF server's response to the commit operation should indicate the success or failure of the commit. You can also verify the success of the commit by reviewing the commit events for the ephemeral database in the system log file.

Action

Review the system log file and display events that match `UI_EPHEMERAL`.

```
user@host> show log messages | match UI_EPHEMERAL
Feb 10 13:20:32  host mgd[5172]: UI_EPHEMERAL_COMMIT: User 'user' has requested commit on 'eph1'
ephemeral database
Feb 10 13:20:32  host mgd[5172]: UI_EPHEMERAL_COMMIT_COMPLETED: commit complete on 'eph1'
ephemeral database
```

Meaning

The `UI_EPHEMERAL_COMMIT_COMPLETED` message tag indicates that the commit operation on the `eph1` instance was successful.

Verify the Configuration Data in the Ephemeral Database Instance

Purpose

Verify that the ephemeral instance has the correct configuration data.

Action

Within the NETCONF session, open the ephemeral database instance and retrieve the configuration.

```
<rpc>
    <open-configuration>
        <ephemeral-instance>eph1</ephemeral-instance>
    </open-configuration>
</rpc>
]]>]]>

<rpc>
    <get-configuration format="text"/>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/25.2R1.9/junos">
<configuration-text xmlns="http://xml.juniper.net/xnm/1.1/xnm">
## Last changed: 2025-10-10 18:46:38 PDT
protocols {
    mpls {
        label-switched-path to-hastings {
            to 192.0.2.1;
        }
    }
}
</configuration-text>
</rpc-reply>
]]>]]>
```

```
<rpc>
    <close-configuration/>
</rpc>
]]>]]>
```

TIP: You can view the configuration data committed to an ephemeral database instance from the CLI by issuing the `show ephemeral-configuration instance` *`instance-name`* operational command.

Troubleshooting

IN THIS SECTION

Troubleshoot Issues When Opening the Ephemeral Instance

Problem

You attempt to open an instance of the ephemeral database, and the server returns only an opening `<rpc-reply>` tag. For example:

```
<rpc>
     <open-configuration>
         <ephemeral-instance>eph1</ephemeral-instance>
     </open-configuration>
</rpc>
]]>]]>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/16.2R2/junos">
```

This issue can occur when another client has the exclusive lock on that instance.

Solution

If another user has an exclusive lock on the ephemeral instance, a client application can issue remote procedure calls (RPCs) to update the ephemeral instance. However, the device does not process the operations on that ephemeral instance until the lock is released. When the lock is released, the server should issue the closing `</rpc-reply>` tag and process any RPCs emitted while the ephemeral instance was locked.

Alternatively, a client application can choose to update a different ephemeral instance, but with the caveat that different ephemeral instances have different priority levels when resolving conflicting configuration statements.

Troubleshoot Operational Issues

Problem

The device does not execute operational changes that should occur as a result of committing certain configuration data to the ephemeral database instance. This occurs even though you have verified that the commit was successful and that the configuration data is present in the configuration for that ephemeral instance.

The operational changes might not occur if the device has another user-defined ephemeral instance that has conflicting configuration data and a higher priority. If the ephemeral instances have conflicting data, statements in an instance with a higher priority override statements in an instance with a lower priority. A user-defined instance of the ephemeral configuration database has higher priority than the default ephemeral database instance, which has higher priority than the static configuration database. If the device has multiple user-defined ephemeral instances in use, the priority is determined by the order in which the instances are listed in the configuration.

Solution

You can verify the configured ephemeral instances and their priority order by issuing the **show configuration system configuration-database ephemeral** operational command on the device. The configuration lists the instances in order from highest to lowest priority. If the device has other instances in use with a higher priority, review the configuration data in those instances to determine if there are conflicting statements. You can also display the merged view of the static and ephemeral configuration databases by issuing the `show ephemeral-configuration merge` command.

If your ephemeral instance has conflicting configuration data and a lower priority than another user-defined ephemeral instance, and the configuration at that hierarchy level should go into effect on the device, you must either delete the conflicting data in the other ephemeral instance or place your configuration data in a higher priority instance.

RELATED DOCUMENTATION

Understanding the Ephemeral Configuration Database | 328

Enable and Configure Instances of the Ephemeral Configuration Database | 346 *ephemeral*

4

PART

Request Operational and Configuration Information Using NETCONF

CHAPTER 13

Request Operational Information Using NETCONF

IN THIS CHAPTER

Request Operational Information Using NETCONF

SUMMARY

A NETCONF client application can use Junos XML request tags to request operational information from Junos devices.

Within a NETCONF session, a client application can request information about the current status of a Junos device. To request operational information, a client application emits the specific request tag element from the Junos XML API that returns the desired information.

Table 19 on page 386 provides examples of request tags, which request the same information as the equivalent CLI command.

Table 19: Examples of Request Tags and Equivalent CLI Command

Request Tag	CLI Command
`<get-interface-information>`	`show interfaces`
`<get-chassis-inventory>`	`show chassis hardware`
`<get-system-inventory>`	`show software information`

You can determine the appropriate Junos XML request tag using multiple methods, including:

- Appending `| display xml rpc` to an operational command in the CLI.
- Using the Junos XML API Explorer - Operational Tags application to search for a command or request tag in a given release.

For example, the following command displays the request tag corresponding to the `show interfaces` command:

```
user@router> show interfaces | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.4R1.9/junos">
    <rpc>
        <get-interface-information>
        </get-interface-information>
    </rpc>
</rpc-reply>
```

To execute an RPC, the client application encloses a request tag in an `<rpc>` element. The syntax depends on whether the corresponding CLI command has any options included.

```
<rpc>
    <!-- If the command does not have options -->
    <operational-request/>

    <!-- If the command has options -->
    <operational-request>
        <!-- tag elements representing the options -->
    </operational-request>
</rpc>
]]>]]>
```

The client application can specify the formatting of the information returned by the NETCONF server. By setting the optional `format` attribute in the opening operational request tag, a client application can specify the format of the response as either XML-tagged format, which is the default, formatted ASCII text, or JavaScript Object Notation (JSON). For more information about specifying the format, see "Specify the Output Format for Operational Information Requests in a NETCONF Session" on page 390.

NOTE: When displaying operational or configuration data that contains characters outside the 7-bit ASCII character set, Junos OS escapes and encodes these character using the equivalent UTF-8 decimal character reference. For more information see "How Character Encoding Works on Juniper Networks Devices" on page 142.

If the client application requests XML output, the NETCONF server encloses its response in the specific response tag element that corresponds to the request tag element, which is then enclosed in an `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <operational-response xmlns="URL-for-DTD">
        <!-- tag elements for the requested information -->
    </operational-response>
</rpc-reply>
]]>]]>
```

For example, if the client application sends the `<get-interface-information>` RPC, the NETCONF server returns the `<interface-information>` response tag.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/23.4R1.9/junos">
<interface-information xmlns="http://xml.juniper.net/junos/23.4R1.9/junos-interface"
junos:style="normal">
<physical-interface>
<name>
ge-0/0/0
</name>
<admin-status junos:format="Enabled">
up
</admin-status>
...
```

For XML format, the opening tag for each operational response includes the `xmlns` attribute. The attribute defines the XML namespace for the enclosed tag elements that do not have a namespace prefix (such as `junos:`). The namespace indicates which Junos XML document type definition (DTD) defines the set of tag elements in the response.

The Junos XML API defines separate DTDs for operational responses from different software modules. For instance, the DTD for interface information is called **junos-interface.dtd** and the DTD for chassis information is called **junos-chassis.dtd**. The division into separate DTDs and XML namespaces means that a tag element with the same name can have distinct functions depending on which DTD it is defined in.

The namespace is a URL of the following form:

```
http://xml.juniper.net/junos/release-code/junos-category
```

where:

- *release-code* is the standard string that represents the Junos OS release that is running on the NETCONF server device.
- *category* specifies the DTD.

If the client application requests the output in formatted ASCII text, the NETCONF server encloses its response in an `<output>` tag element, which is enclosed in an `<rpc-reply>` tag.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <output>
        operational-response
    </output>
</rpc-reply>
]]>]]>
```

If the client application requests the output in JSON format, the NETCONF server encloses the JSON data in the `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    operational-response
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Understanding the Request Procedure in a NETCONF Session | **39**

Specify the Output Format for Operational Information Requests in a NETCONF Session | **390**

Request Configuration Data Using NETCONF | **400**

Specify the Output Format for Operational Information Requests in a NETCONF Session

SUMMARY

A NETCONF client application can include the `format` attribute in Junos XML request tags to specify the output format for operational information requests on Junos devices.

In a NETCONF session, to request information about a Junos device, a client application emits an `<rpc>` element that encloses a Junos XML request tag element. To request that the NETCONF server return the output in a specific format, the client application includes the optional `format` attribute in the opening operational request tag. The application can request output in formatted ASCII text, JavaScript Object Notation (JSON), or XML-tagged format. The syntax is as follows:

```
<rpc>
    <operational-request format="(ascii | json | json-minified | text | xml | xml-minified)">
        <!-- tag elements for options -->
    </operational-request>
</rpc>
```

Table 20 on page 390 describes the available formats. Minified formats remove characters that are not required for computer processing, for example, spaces, tabs, and newlines. Minified formats decrease the size of the data, and as a result, can reduce transport costs and data delivery and processing times.

Table 20: Operational RPC Output Formats

`format` Attribute Value	Description
`ascii`	Formatted ASCII text
`json`	JSON format
`json-minified`	JSON format with unnecessary spaces, tabs, and newlines removed
`text`	Formatted ASCII text

Table 20: Operational RPC Output Formats *(Continued)*

`format` Attribute Value	Description
`xml`	Junos XML-tagged format
`xml-minified`	Junos XML-tagged format with unnecessary spaces, tabs, and newlines removed

XML Format

By default, the NETCONF server returns operational information in XML format. If a client application sets the `format` attribute to `xml` or omits the `format` attribute, the server returns the response in XML. The following example requests information for the ge-0/3/0 interface and omits the `format` attribute.

```
<rpc>
     <get-interface-information>
          <brief/>
          <interface-name>ge-0/3/0</interface-name>
     </get-interface-information>
</rpc>
```

The NETCONF server returns the information in XML. The output is identical to the CLI output when you append the `| display xml` filter to the operational mode command.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
<interface-information
 xmlns="http://xml.juniper.net/junos/25.2R1.9/junos-interface" junos:style="brief">
        <physical-interface>
            <name>ge-0/3/0</name>
            <admin-status junos:format="Enabled">up</admin-status>
            <oper-status>down</oper-status>
            <link-level-type>Ethernet</link-level-type>
            <mtu>1514</mtu>
            <source-filtering>disabled</source-filtering>
            <speed>1000mbps</speed>
            <bpdu-error>none</bpdu-error>
            <l2pt-error>none</l2pt-error>
            <loopback>disabled</loopback>
            <if-flow-control>enabled</if-flow-control>
            <if-auto-negotiation>enabled</if-auto-negotiation>
            <if-remote-fault>online</if-remote-fault>
            <if-device-flags>
                <ifdf-present/>
                <ifdf-running/>
                <ifdf-down/>
            </if-device-flags>
            <if-config-flags>
                <iff-hardware-down/>
                <iff-snmp-traps/>
                <internal-flags>0x4000</internal-flags>
            </if-config-flags>
            <if-media-flags>
                <ifmf-none/>
            </if-media-flags>
        </physical-interface>
    </interface-information>
</rpc-reply>
```

Operational command RPCs also support returning XML output in minified format, which omits unnecessary spaces, tabs, and newlines. To request minified XML output in supported releases, include the `format="xml-minified"` attribute in the opening request tag. For example:

```
<rpc>
     <get-interface-information format="xml-minified">
          <brief/>
          <interface-name>ge-0/3/0</interface-name>
     </get-interface-information>
</rpc>
```

The NETCONF server returns the information in minified XML format.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/25.2R1.9/junos">
<interface-information xmlns="http://xml.juniper.net/junos/25.2R1.9/junos-interface"
junos:style="brief"><physical-interface><name>ge-0/3/0</name><admin-status
junos:format="Enabled">up</admin-status><oper-status>down</oper-status><link-level-
type>Ethernet</link-level-type><mtu>1514</mtu><source-filtering>disabled</source-
filtering><speed>1000mbps</speed><bpdu-error>none</bpdu-error><l2pt-error>none</l2pt-
error><loopback>disabled</loopback><if-flow-control>enabled</if-flow-control><if-auto-
negotiation>enabled</if-auto-negotiation><if-remote-fault>online</if-remote-fault><if-device-
```

```
flags><ifdf-present/><ifdf-running/><ifdf-down/></if-device-flags><if-config-flags><iff-hardware-
down/><iff-snmp-traps/><internal-flags>0x4000</internal-flags></if-config-flags><if-media-
flags><ifmf-none/></if-media-flags></physical-interface></interface-information></rpc-reply>
```

ASCII Format

To request that the NETCONF server return operational information as formatted ASCII text, the client application includes the `format="text"` or `format="ascii"` attribute in the opening request tag.

```
<rpc>
     <get-interface-information format="(text | ascii)">
          <brief/>
          <interface-name>ge-0/3/0</interface-name>
     </get-interface-information>
</rpc>
```

When the client application includes the `format="text"` or `format="ascii"` attribute in the request tag, the NETCONF server formats the reply as ASCII text and encloses it in an `<output>` tag element. The `format="text"` and `format="ascii"` attributes produce identical output.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
<output>
Physical interface: ge-0/3/0, Enabled, Physical link is Down
  Link-level type: Ethernet, MTU: 1514, Speed: 1000mbps, Loopback: Disabled,
  Source filtering: Disabled, Flow control: Enabled, Auto-negotiation: Enabled,
  Remote fault: Online
  Device flags   : Present Running Down
  Interface flags: Hardware-Down SNMP-Traps Internal: 0x4000
  Link flags     : None
</output>
</rpc-reply>
```

The following example shows the equivalent operational mode command executed in the CLI:

```
user@host> show interfaces ge-0/3/0 brief
Physical interface: ge-0/3/0, Enabled, Physical link is Down
  Link-level type: Ethernet, MTU: 1514, Speed: 1000mbps, Loopback: Disabled, Source filtering:
Disabled,
  Flow control: Enabled, Auto-negotiation: Enabled, Remote fault: Online
  Device flags   : Present Running Down
```

```
    Interface flags: Hardware-Down SNMP-Traps Internal: 0x4000
    Link flags      : None
```

The formatted ASCII text returned by the NETCONF server is identical to the CLI output except in cases where the output includes disallowed characters. Disallowed characters include '<' (less-than sign), '>' (greater-than sign), and '&' (ampersand). The NETCONF server substitutes these characters with the equivalent predefined entity reference of '&lt;', '&gt;', and '&amp;' respectively.

If the Junos XML API does not define a response tag element for the type of output requested by a client application, the NETCONF server returns the reply as formatted ASCII text enclosed in an `<output>` tag element, even if XML-tagged output is requested.

NOTE: The content and formatting of data within an `<output>` tag element are subject to change, so client applications must not depend on them.

JSON Format

A client application can request operational and configuration data in JSON format. To request that the NETCONF server return operational information in JSON format, the client application includes the `format="json"` attribute in the opening request tag.

```
<rpc>
     <get-interface-information format="json">
          <brief/>
          <interface-name>cbp0</interface-name>
     </get-interface-information>
</rpc>
```

When the client application includes the `format="json"` attribute in the request tag, the NETCONF server returns JSON data.

```

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
{
 "interface-information" : [
 {
    "physical-interface" : [
    {
        "name" : [
        {
```

```
    "data" : "cbp0"
}
],
"admin-status" : [
{
    "data" : "up",
    "attributes" : {"junos:format" : "Enabled"}
}
],
"oper-status" : [
{
    "data" : "up"
}
],
"if-type" : [
{
    "data" : "Ethernet"
}
],
"link-level-type" : [
{
    "data" : "Ethernet"
}
],
"mtu" : [
{
    "data" : "9192"
}
],
"speed" : [
{
    "data" : "Unspecified"
}
],
"clocking" : [
{
    "data" : "Unspecified"
}
],
"if-device-flags" : [
{
    "ifdf-present" : [
    {
```

```
                "data" : [null]
            }
            ],
            "ifdf-running" : [
            {
                "data" : [null]
            }
            ]
        }
        ],
        "ifd-specific-config-flags" : [
        {
        }
        ],
        "if-config-flags" : [
        {
            "iff-snmp-traps" : [
            {
                "data" : [null]
            }
            ]
        }
        ]
    }
    ]
 }
 ]
}
</rpc-reply>
```

By default, Junos devices emit JSON-formatted state data in non-compact format, which emits all objects as JSON arrays. In Junos OS Release 24.2 and earlier and Junos OS Evolved Release 24.2 and earlier, Junos devices support emitting the device's operational state in compact JSON format. Compact JSON format emits JSON arrays only for objects that have multiple values. To configure the device to emit compact JSON format, configure the `compact` statement at the `[edit system export-format state-data json]` hierarchy level.

NETCONF clients can also request operational command RPC output in minified JSON format, which omits unnecessary spaces, tabs, and newlines. To request minified JSON output in supported releases, include the `format="json-minified"` attribute in the opening request tag. For example:

```
<rpc>
     <get-interface-information format="json-minified">
          <brief/>
          <interface-name>cbp0</interface-name>
     </get-interface-information>
</rpc>
```

The NETCONF server returns the information in minified JSON format.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/25.2R1.9/junos">
{"interface-information":[{"physical-interface":[{"name":[{"data":"cbp0"}],"admin-status":
[{"data":"up","attributes":{"junos:format":"Enabled"}}],"oper-status":[{"data":"up"}],"if-type":
[{"data":"Ethernet"}],"link-level-type":[{"data":"Ethernet"}],"mtu":[{"data":"9192"}],"speed":
[{"data":"Unspecified"}],"clocking":[{"data":"Unspecified"}],"if-device-flags":[{"ifdf-present":
[{"data":[null]}],"ifdf-running":[{"data":[null]}]}],"ifd-specific-config-flags":[{}],"if-config-
flags":[{"iff-snmp-traps":[{"data":[null]}]}]}]}]}</rpc-reply>
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
24.4R1 & 24.4R1-EVO	Starting in Junos OS Release 24.4R1 and Junos OS Evolved Release 24.4R1, we've deprecated the `compact` statement at the `[edit system export-format state-data json]` hierarchy level.

CHAPTER 14

Request Configuration Information Using NETCONF

IN THIS CHAPTER

Request the Committed Configuration and Device State Using NETCONF

In a NETCONF session with a device running Junos OS, to request the most recently committed configuration and the device state information for a routing, switching, or security platform, a client application encloses the `<get>` tag in an `<rpc>` tag element. By including the `<filter>` tag element and appropriate child tag elements, the application can request specific portions of the configuration. If the

`<filter>` element is omitted, the server returns the entire configuration. The optional `format` attribute specifies the return format for the configuration data.

```
<rpc>
    <get [format="(json | set | text | xml)"]>
        <filter type="subtree">
             <!-- tag elements representing the configuration elements to return -->
         </filter>
    </get>
</rpc>
]]>]]>
```

The `type="subtree"` attribute in the opening `<filter>` tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

The NETCONF server encloses its reply in the `<rpc-reply>` and `<data>` tag elements. Within the `<data>` element, the configuration data is enclosed in the `<configuration>`, `<configuration-text>`, `<configuration-set>`, or `<configuration-json>` element depending on the requested format, and the device information is enclosed in the `<database-status-information>` element. The server includes attributes in the opening `<configuration>` tag that indicate the XML namespace for the enclosed tag elements and when the configuration was last changed or committed. For example:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration xmlns="URL" junos:changed-seconds="seconds" junos:changed-
localtime="time">
            <!-- configuration data -->
        </configuration>
        <database-status-information>
            <database-status>
                <user>user</user>
                <terminal></terminal>
                <pid>pid</pid>
                <start-time junos:seconds="1416956595">2014-11-25 15:03:15 PST</start-time>
                <edit-path></edit-path>
            </database-status>
        </database-status-information>
    </data>
</rpc-reply>
]]>]]>
```

If there is no configuration data in the requested hierarchy, the RPC reply contains an empty `<configuration>` tag inside the `<data>` element unless the `rfc-compliant` statement is configured, in which case the `<configuration>` tag is omitted.

RELATED DOCUMENTATION

<get> | 195

Request Configuration Data Using NETCONF | 400

Request Configuration Data Using NETCONF

In a NETCONF session with a device running Junos OS, to request configuration data for a routing, switching, or security platform, a client application encloses the `<get-config>`, `<source>`, and `<filter>` tag elements in an `<rpc>` tag element. By including the appropriate child tag element in the `<source>` tag element, the client application requests information from the active configuration or from the candidate configuration or open configuration database. By including the appropriate child tag elements in the `<filter>` tag element, the application can request the entire configuration or specific portions of the configuration.

```
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
            <( candidate | running )/>
        </source>
        <filter type="subtree">
             <!-- tag elements representing the configuration elements to return -->
         </filter>
    </get-config>
</rpc>
]]>]]>
```

The `type="subtree"` attribute in the opening `<filter>` tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

NOTE: If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-config>` operation, setting the source to `<candidate/>` retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

NOTE: If the client application locks the candidate configuration before making requests, it needs to unlock it after making its read requests. Other users and applications cannot change the configuration while it remains locked.

The NETCONF server encloses its reply in `<rpc-reply>`, `<data>`, and `<configuration>` tag elements. It includes attributes in the opening `<configuration>` tag that indicate the XML namespace for the enclosed tag elements and when the configuration was last changed or committed. For information about the attributes of the `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
     <data>
         <configuration attributes>
            <!-- JUNOS XML tag elements representing configuration elements -->
         </configuration>
    </data>
</rpc-reply>
]]>]]>
```

If a Junos XML tag element is returned within an `<undocumented>` tag element, the corresponding configuration element is not documented in the Junos OS configuration guides or officially supported by Juniper Networks. Most often, the enclosed element is used for debugging only by support personnel. In a smaller number of cases, the element is no longer supported or has been moved to another area of the configuration hierarchy, but appears in the current location for backward compatibility.

NOTE: When displaying operational or configuration data that contains characters outside the 7-bit ASCII character set, Junos OS escapes and encodes these character using the equivalent UTF-8 decimal character reference. For more information see "How Character Encoding Works on Juniper Networks Devices" on page 142.

Client applications can also request other configuration-related information, including an XML schema representation of the configuration hierarchy or information about previously committed configurations.

RELATED DOCUMENTATION

Request Operational Information Using NETCONF | 386

Specify the Source for Configuration Information Requests Using NETCONF

In a NETCONF session with a device running Junos OS, to request information from the candidate configuration or open configuration database, a client application includes the `<source>` element and `<candidate/>` tag within the `<rpc>` and `<get-config>` tag elements.

```
<rpc>
    <get-config>
        <source>
            <candidate/>
        </source>
        <filter>
            <!-- tag elements representing the configuration elements to return -->
        </filter>
     </get-config>
</rpc>
]]>]]>
```

NOTE: If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-config>` operation, setting the source to `<candidate/>` retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

To request information from the active configuration—the one most recently committed on the device—a client application includes the `<source>` tag element and `<running/>` tag enclosed within the `<rpc>` and `<get-config>` tag elements.

```
<rpc>
    <get-config>
        <source>
            <running/>
        </source>
        <filter>
            <!-- tag elements representing the configuration elements to return -->
        </filter>
     </get-config>
</rpc>
]]>]]>
```

NOTE: If a client application is requesting the entire configuration, it omits the `<filter>` tag element.

The NETCONF server encloses its reply in `<rpc-reply>`, `<data>`, and `<configuration>` tag elements. In the opening `<configuration>` tag, it includes the `xmlns` attribute to specify the namespace for the enclosed tag elements.

When returning information from the candidate configuration or open configuration database, the NETCONF server includes attributes that indicate when the configuration last changed (they appear on multiple lines here only for legibility).

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration xmlns="URL" junos:changed-seconds="seconds" \
            junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ">
             <!-- Junos XML tag elements representing the configuration -->
          </configuration>
    </data>
</rpc-reply>
]>]]>
```

`junos:changed-localtime` represents the time of the last change as the date and time in the device’s local time zone.

`junos:changed-seconds` represents the time of the last change as the number of seconds since midnight on 1 January 1970.

When returning information from the active configuration, the NETCONF server includes attributes that indicate when the configuration was committed (they appear on multiple lines here only for legibility).

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration xmlns="URL" junos:commit-seconds="seconds" \
             junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
               junos:commit-user="username">
              <!-- Junos XML tag elements representing the configuration -->
          </configuration>
     </data>
</rpc-reply>
]]>]]>
```

`junos:commit-localtime` represents the commit time as the date and time in the device's local time zone.

`junos:commit-seconds` represents the commit time as the number of seconds since midnight on 1 January 1970.

`junos:commit-user` specifies the Junos OS username of the user who requested the commit operation.

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | 400

<get-config> | 198

Specify the Scope of Configuration Information to Return in a NETCONF Response

In a NETCONF session with a device running Junos OS, a client application can request the entire configuration or specific portions of the configuration by including the appropriate child tag elements in the `<filter>` tag element within the `<rpc>` and `<get-config>` tag elements.

```
<rpc>
    <get-config>
        <source>
            ( <candidate/> | <running/> )
        </source>
        <filter type="subtree">
            <!-- tag elements representing the configuration elements to return -->
        </filter>
    </get-config>
</rpc>
]]>]]>
```

The `type="subtree"` attribute in the opening `<filter>` tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

For information about requesting different amounts of configuration information, see the following topics:

- "Request the Complete Configuration Using NETCONF" on page 405
- "Request a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF" on page 407
- "Request All Configuration Objects of a Specified Type Using NETCONF" on page 410
- "Request Identifiers for Configuration Objects of a Specified Type Using NETCONF" on page 413
- "Request A Specific Configuration Object Using NETCONF" on page 416
- "Request Specific Child Tags for a Configuration Object Using NETCONF" on page 419
- "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | **400**

Specify the Source for Configuration Information Requests Using NETCONF | **402**

Request the Complete Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, to request the entire candidate configuration or the complete configuration in the open configuration database, a client application encloses `<get-config>` and `<source>` tag elements and the `<candidate/>` tag in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <candidate/>
        </source>
    </get-config>
</rpc>
]]>]]>
```

NOTE: If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-config>` operation, setting the source to `<candidate/>` retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

To request the entire active configuration, a client application encloses `<get-config>` and `<source>` tag elements and the `<running/>` tag in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <running/>
        </source>
    </get-config>
</rpc>
]]>]]>
```

The NETCONF server encloses its reply in `<rpc-reply>`, `<data>`, and `<configuration>` tag elements. For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
         <configuration attributes>
             <!-- Junos XML tag elements representing the configuration -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | 400

Specify the Source for Configuration Information Requests Using NETCONF | 402

Specify the Scope of Configuration Information to Return in a NETCONF Response | 404

Retrieve the Rescue Configuration Using NETCONF | 432

Request a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF

In a NETCONF session with a device running Junos OS, to request complete information about all child configuration elements at a hierarchy level or in a container object that does not have an identifier, a client application emits a `<filter>` tag element that encloses the tag elements representing all levels in the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level of the level or container object, which is represented by an empty tag. The entire request is enclosed in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
         </source>
        <filter type="subtree">
            <configuration>
                <!-- opening tags for each parent of the requested level -->
                    <level-or-container/>
                <!-- closing tags for each parent of the requested level -->
            </configuration>
        </filter>
     </get-config>
</rpc>
]]>]]>
```

For information about the `<source>` tag element, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

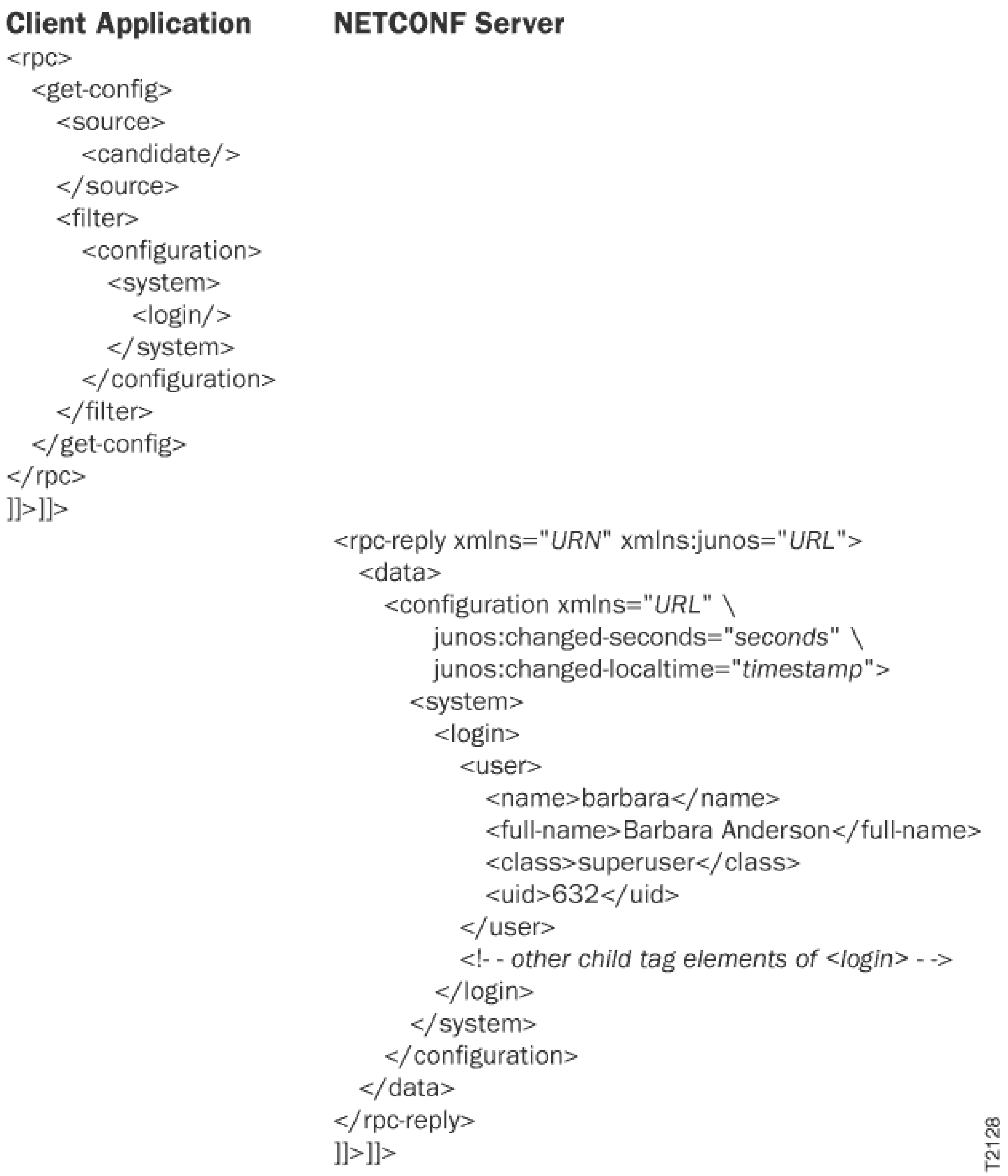
The NETCONF server returns the requested section of the configuration in `<data>` and `<rpc-reply>` tag elements. For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration attributes>
            <!-- opening tags for each parent of the level -->
```

```
                 <level-or-container>
                      <!-- child tag elements of the level or container -->
                 </level-or-container>
             <!-- closing tags for each parent of the level -->
         </configuration>
     </data>
 </rpc-reply>
 ]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-config>` tag element. For more information, see "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424.

The following example shows how to request the contents of the `[edit system login]` hierarchy level in the candidate configuration.



Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <system>
          <login/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <system>
        <login>
          <user>
            <name>barbara</name>
            <full-name>Barbara Anderson</full-name>
            <class>superuser</class>
            <uid>632</uid>
          </user>
          <!- - other child tag elements of <login> - ->
        </login>
      </system>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | 400

Specify the Scope of Configuration Information to Return in a NETCONF Response | 404

Request All Configuration Objects of a Specified Type Using NETCONF

In a NETCONF session with a device running Junos OS, to request information about all configuration objects of a specified type in a hierarchy level, a client application emits a `<filter>` tag element that encloses the tag elements representing all levels in the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level for the object type. An empty tag returns all configuration objects of the requested object type and all child tags for each object. To return only specific child tags for the configuration objects, enclose the desired child tags in the opening and closing tags of the object. The entire request is enclosed in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
           <configuration>
              <!-- opening tags for each parent of the requested object type -->
                 <object-type>
                     <!-- optionally select specific child tags -->
                 </object-type>
              <!-- closing tags for each parent of the requested object type -->
           </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
```

For information about the `<source>` tag element, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

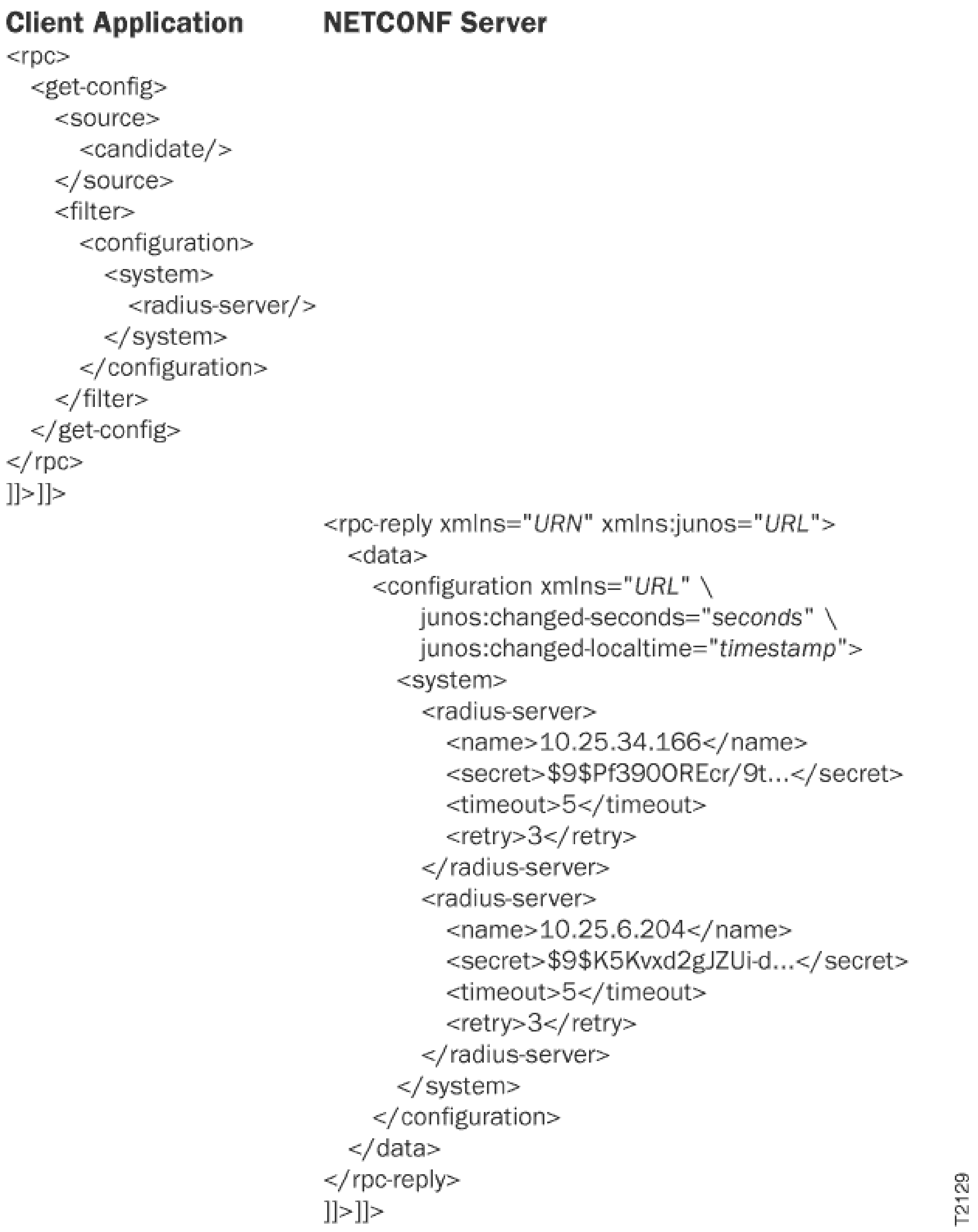
This type of request is useful when the object's parent hierarchy level has more than one type of child object. If the requested object is the only child type that can occur in its parent hierarchy level, then this type of request yields the same output as a request for the complete parent hierarchy, which is described in "Request a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF" on page 407.

The NETCONF server returns the requested objects in `<data>` and `<rpc-reply>` tag elements. For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration attributes>
            <!-- opening tags for each parent of the object type -->
                <first-object>
                    <!-- child tag elements for the first object -->
                </first-object>
                <second-object>
                    <!-- child tag elements for the second object -->
                </second-object>
                <!-- additional instances of the object -->
            <!-- closing tags for each parent of the object type -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-config>` tag element. For more information, see "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424.

The following example shows how to request complete information about all `radius-server` objects at the `[edit system]` hierarchy level in the candidate configuration.



Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <system>
          <radius-server/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <system>
        <radius-server>
          <name>10.25.34.166</name>
          <secret>$9$Pf390OREcr/9t...</secret>
          <timeout>5</timeout>
          <retry>3</retry>
        </radius-server>
        <radius-server>
          <name>10.25.6.204</name>
          <secret>$9$K5Kvxd2gJZUi-d...</secret>
          <timeout>5</timeout>
          <retry>3</retry>
        </radius-server>
      </system>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

T2129

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | **400**

Specify the Source for Configuration Information Requests Using NETCONF | **402**

Specify the Scope of Configuration Information to Return in a NETCONF Response | **404**

Request Identifiers for Configuration Objects of a Specified Type Using NETCONF | **413**

Request Identifiers for Configuration Objects of a Specified Type Using NETCONF

In a NETCONF session with a device running Junos OS, to request output that shows only the identifier for each configuration object of a specific type in a hierarchy, a client application emits a `<filter>` tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level for the object type. The object type is represented by its container tag element enclosing an empty `<name/>` tag. (The `<name>` tag element can always be used, even if the actual identifier tag element has a different name. The actual name is also valid.) The entire request is enclosed in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
            <configuration>
                 <!-- opening tags for each parent of the object type -->
                    <object-type>
                        <name/>
                    </object-type>
                 <!-- closing tags for each parent of the object type -->
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
```

For information about the `<source>` tag element, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

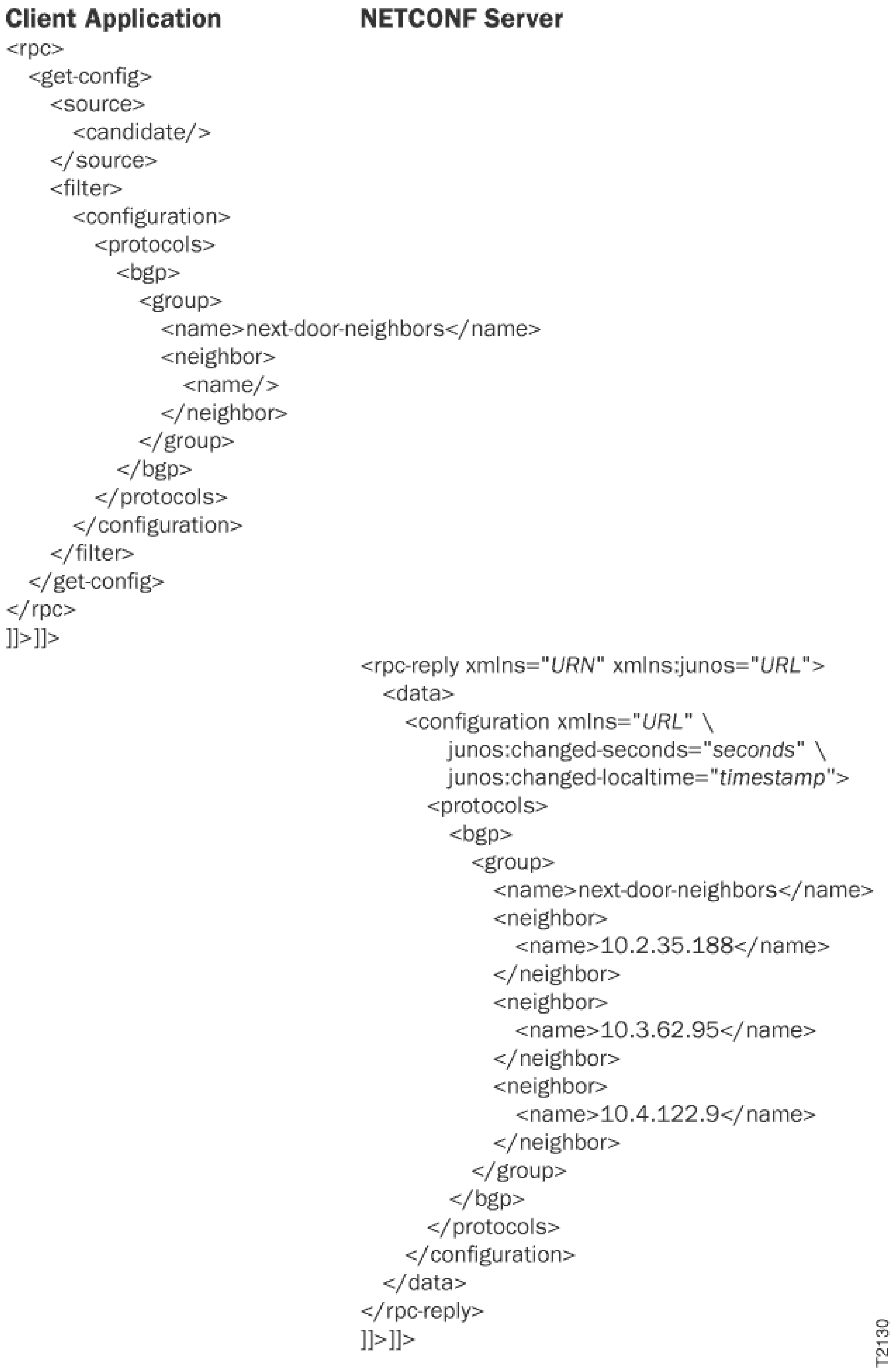
NOTE: You cannot request only identifiers for object types that have multiple identifiers. However, for many such objects the identifiers are the only child tag elements, so requesting complete information yields the same output as requesting only identifiers. For instructions, see "Request All Configuration Objects of a Specified Type Using NETCONF" on page 410.

The NETCONF server returns the requested objects in `<data>` and `<rpc-reply>` tag elements (here, objects for which the identifier tag element is called `<name>`). For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration attributes>
            <!-- opening tags for each parent of the object type -->
                <first-object>
                    <name>identifier-for-first-object</name>
                </first-object>
                <second-object>
                    <name>identifier-for-second-object</name>
                </second-object>
                <!-- additional objects -->
            <!-- closing tags for each parent of the object type -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-config>` tag element. For more information, see "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424.

The following example shows how to request the identifier for each BGP neighbor configured at the `[edit protocols bgp group next-door-neighbors]` hierarchy level in the candidate configuration.



```
Client Application                 NETCONF Server
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <protocols>
          <bgp>
            <group>
              <name>next-door-neighbors</name>
              <neighbor>
                <name/>
              </neighbor>
            </group>
          </bgp>
        </protocols>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
                                   <rpc-reply xmlns="URN" xmlns:junos="URL">
                                     <data>
                                       <configuration xmlns="URL" \
                                           junos:changed-seconds="seconds" \
                                           junos:changed-localtime="timestamp">
                                         <protocols>
                                           <bgp>
                                             <group>
                                               <name>next-door-neighbors</name>
                                               <neighbor>
                                                 <name>10.2.35.188</name>
                                               </neighbor>
                                               <neighbor>
                                                 <name>10.3.62.95</name>
                                               </neighbor>
                                               <neighbor>
                                                 <name>10.4.122.9</name>
                                               </neighbor>
                                             </group>
                                           </bgp>
                                         </protocols>
                                       </configuration>
                                     </data>
                                   </rpc-reply>
                                   ]]>]]>
```

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | 400

Specify the Source for Configuration Information Requests Using NETCONF | 402

Specify the Scope of Configuration Information to Return in a NETCONF Response | 404

Request All Configuration Objects of a Specified Type Using NETCONF | 410

Request A Specific Configuration Object Using NETCONF

In a NETCONF session with a device running Junos OS, to request complete information about a specific configuration object, a client application emits a `<filter>` tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level for the object.

To represent the requested object, the application emits only the container tag element and each of its identifier tag elements, complete with identifier value, for the object. For objects with a single identifier, the `<name>` tag element can always be used, even if the actual identifier tag element has a different name. The actual name is also valid. For objects with multiple identifiers, the actual names of the identifier tag elements must be used. To verify the name of each of the identifiers for a configuration object, see the *Junos XML API Configuration Developer Reference*. The entire request is enclosed in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <!--tag specifying the source configuration -->
         </source>
         <filter type="subtree">
            <configuration>
                <!-- opening tags for each parent of the object -->
                    <object>
                        <name>identifier</name>
                    </object>
                <!-- closing tags for each parent of the object -->
            </configuration>
        </filter >
    </get-config>
</rpc>
]]>]]>
```

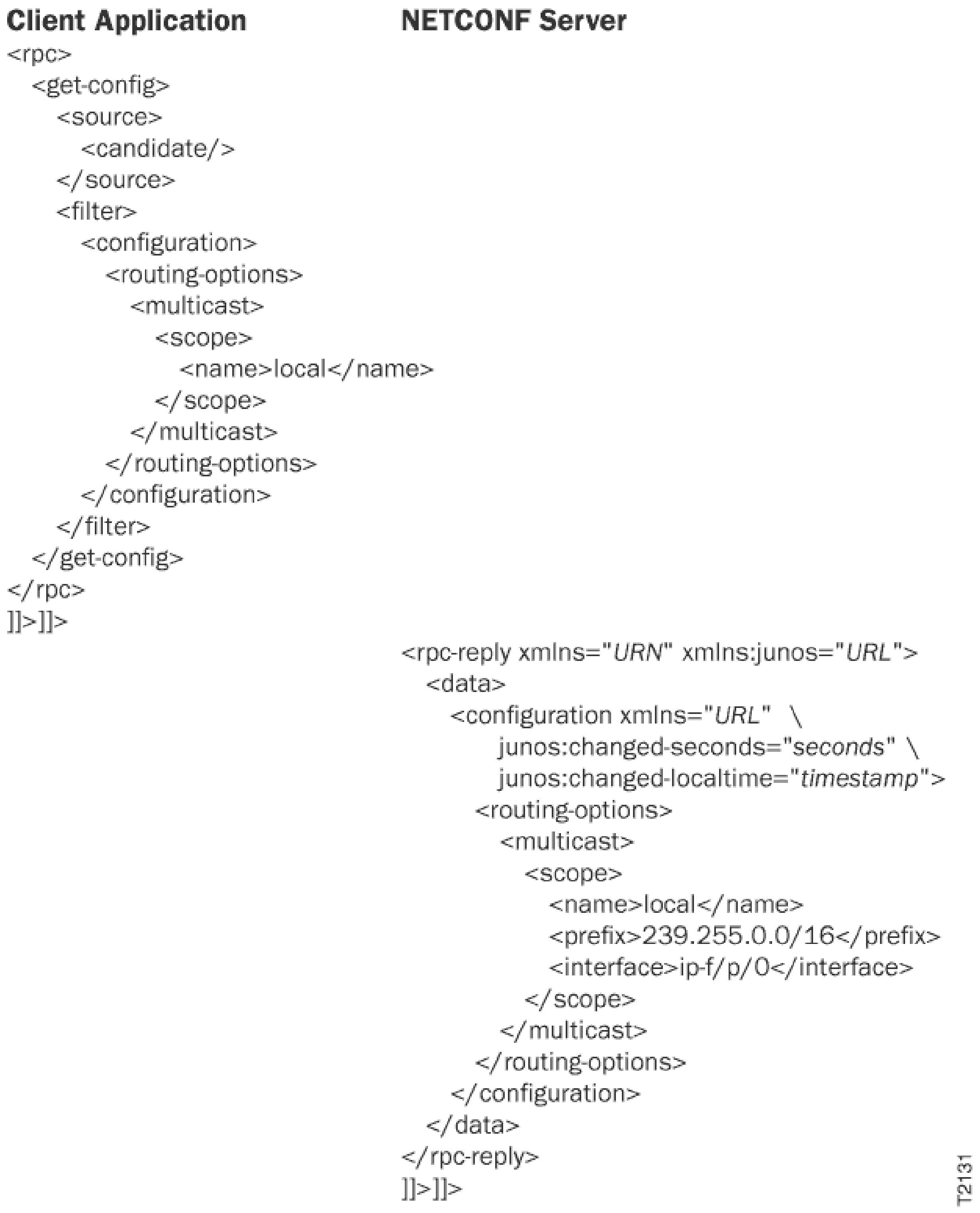
For information about the <source> tag element, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

The NETCONF server returns the requested object in <data> and <rpc-reply> tag elements (here, an object for which the identifier tag element is called <name>). For information about the attributes in the opening <configuration> tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration attributes>
            <!-- opening tags for each parent of the object -->
                 <object>
                    <name>identifier</name>
                    <!-- other child tag elements of the object -->
                </object>
            <!-- closing tags for each parent of the object -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same <get-config> tag element. For more information, see "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424.

The following example shows how to request the contents of one multicasting scope called local, which is at the [edit routing-options multicast] hierarchy level in the candidate configuration. To specify the desired object, the client application emits the <name>local</name> identifier tag element as the innermost tag element.



Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <routing-options>
          <multicast>
            <scope>
              <name>local</name>
            </scope>
          </multicast>
        </routing-options>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL"  \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <routing-options>
        <multicast>
          <scope>
            <name>local</name>
            <prefix>239.255.0.0/16</prefix>
            <interface>ip-f/p/0</interface>
          </scope>
        </multicast>
      </routing-options>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Request Specific Child Tags for a Configuration Object Using NETCONF

In a NETCONF session with a device running Junos OS, to request specific child tag elements and descendents for configuration objects, a client application emits a `<filter>` tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level for the object. To represent the requested object, the application emits its container tag element. To request a specific configuration object, include the identifier tag element. For objects with a single identifier, the `<name>` tag element can always be used, even if the actual identifier tag element has a different name. The actual name is also valid. For objects with multiple identifiers, the actual names of the identifier tag elements must be used. If you omit the identifier tag element, the server returns the child tags for all configuration objects of that type. To select specific child tags, the client application emits all desired child tag elements and descendents within the container tag element. The entire request is enclosed in an `<rpc>` tag element:

```
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
         </source>
        <filter type="subtree">
            <configuration>
                <!-- opening tags for each parent of the object -->
                    <object>
                        <name>identifier</name>
                        <first-child/>
                        <second-child/>
                        <third-child>
                            <!--tags for descendents-->
                        </third-child>
                        <!-- tag for each additional child to return -->
                    </object>
                <!-- closing tags for each parent of the object -->
            </configuration>
        </filter>
    </get-config>

</rpc>
]]>]]>
```

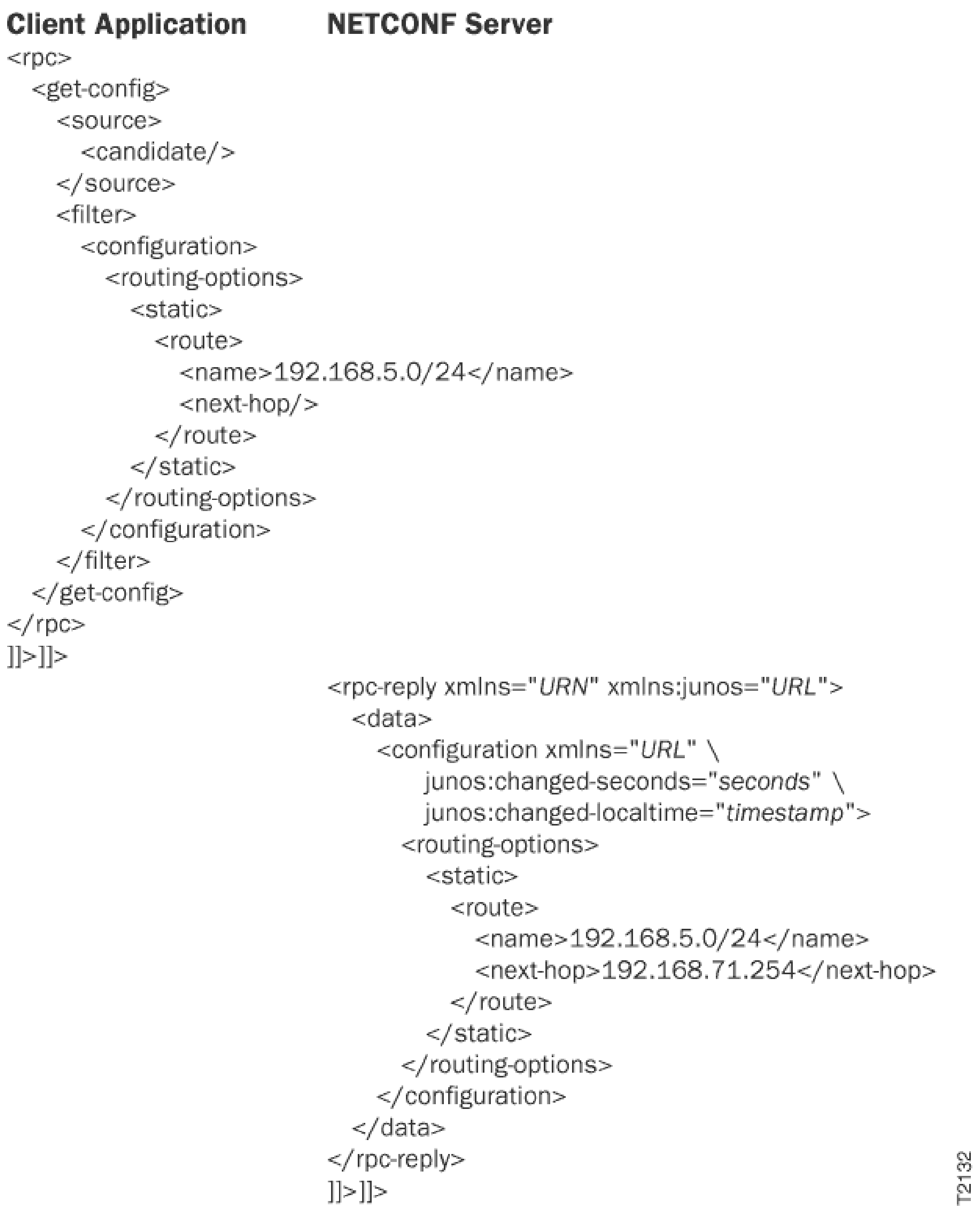
For information about the `<source>` tag element, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

The NETCONF server returns the requested children of the object in `<data>` and `<rpc-reply>` tag elements (here, an object for which the identifier tag element is called `<name>`). For information about the attributes in the opening `<configuration>` tag, see "Specify the Source for Configuration Information Requests Using NETCONF" on page 402.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <data>
        <configuration attributes>
            <!-- opening tags for each parent of the object -->
                <object>
                    <name>identifier</name>
                    <!-- requested child tags -->
                </object>
            <!-- closing tags for each parent of the object -->
        </configuration>
    </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-config>` tag element. For more information, see "Request Multiple Configuration Elements Simultaneously Using NETCONF" on page 424.

The following example shows how to request only the address of the next-hop device for the 192.168.5.0/24 route at the `[edit routing-options static]` hierarchy level in the candidate configuration.



Client Application **NETCONF Server**

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <routing-options>
          <static>
            <route>
              <name>192.168.5.0/24</name>
              <next-hop/>
            </route>
          </static>
        </routing-options>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
                          <rpc-reply xmlns="URN" xmlns:junos="URL">
                            <data>
                              <configuration xmlns="URL" \
                                  junos:changed-seconds="seconds" \
                                  junos:changed-localtime="timestamp">
                                <routing-options>
                                  <static>
                                    <route>
                                      <name>192.168.5.0/24</name>
                                      <next-hop>192.168.71.254</next-hop>
                                    </route>
                                  </static>
                                </routing-options>
                              </configuration>
                            </data>
                          </rpc-reply>
                          ]]>]]>
```

The following example shows how to request the addresses for all logical interfaces configured for each physical interface within the groups hierarchy level of the candidate configuration.

```
<rpc>
  <get-config>
      <source>
        <candidate/>
      </source>
      <filter type="subtree">
        <configuration>
          <groups>
            <interfaces>
              <interface>
                <unit>
                  <family>
                    <inet>
                      <address/>
                    </inet>
                  </family>
                </unit>
              </interface>
            </interfaces>
          </groups>
        </configuration>
      </filter>
  </get-config>
</rpc>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" junos:commit-seconds=seconds junos:commit-localtime="timestamp"
junos:commit-user="user">
      <groups>
        <name>re0</name>
        <interfaces>
          <interface>
            <name>lo0</name>
            <unit>
              <name>0</name>
              <family>
                <inet>
```

```
                  <address>
                    <name>127.0.0.1/32</name>
                  </address>
                </inet>
              </family>
            </unit>
          </interface>
          <interface>
            <name>em0</name>
            <unit>
              <name>0</name>
              <family>
                <inet>
                  <address>
                    <name>198.51.100.1/24</name>
                  </address>
                  <address>
                    <name>198.51.100.11/24</name>
                  </address>
                </inet>
              </family>
            </unit>
          </interface>
        </interfaces>
      </groups>
    </configuration>
  </data>
</rpc-reply>
```

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | **400**

Specify the Source for Configuration Information Requests Using NETCONF | **402**

Specify the Scope of Configuration Information to Return in a NETCONF Response | **404**

Request A Specific Configuration Object Using NETCONF | **416**

Request Multiple Configuration Elements Simultaneously Using NETCONF | **424**

Request Multiple Configuration Elements Simultaneously Using NETCONF

In a NETCONF session with a device running Junos OS, a client application can request multiple configuration elements of the same type or different types within a `<get-config>` tag element. The request includes only one `<filter>` and `<configuration>` tag element (the NETCONF server returns an error if there is more than one of each).

If two requested objects have the same parent hierarchy level, the client can either include both requests within one parent tag element, or repeat the parent tag element for each request. For example, at the `[edit system]` hierarchy level the client can request the list of configured services and the identifier tag element for RADIUS servers in either of the following two ways:

```
<!-- both requests in one <system> tag element -->
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
            <configuration>
                <system>
                    <services/>
                    <radius-server>
                        <name/>
                    </radius-server>
                </system>
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>

<!-- separate <system> tag element for each element -->
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
            <configuration>
```

```
                <system>
                    <services/>
                </system>
                <system>
                    <radius-server>
                        <name/>
                    </radius-server>
                </system>
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
```

The client can combine requests for any of the following types of information:

- "Request a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF" on page 407
- "Request All Configuration Objects of a Specified Type Using NETCONF" on page 410
- "Request Identifiers for Configuration Objects of a Specified Type Using NETCONF" on page 413
- "Request A Specific Configuration Object Using NETCONF" on page 416
- "Request Specific Child Tags for a Configuration Object Using NETCONF" on page 419

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | **400**

Specify the Source for Configuration Information Requests Using NETCONF | **402**

Specify the Scope of Configuration Information to Return in a NETCONF Response | **404**

Retrieve a Previous (Rollback) Configuration Using NETCONF

SUMMARY

A NETCONF client application can use the `<get-rollback-information>` request tag to retrieve a previously committed configuration.

Junos OS and Junos OS Evolved store a copy of the most recently committed configuration and up to 49 previous configurations, depending on the platform. When you successfully commit a configuration, the device associates that configuration with a rollback index, where the most recently committed configuration has rollback index 0. The rollback index for a committed configuration increments with each commit. When you request a previously committed configuration, you can reference the configuration by its current rollback index.

A NETCONF client application can retrieve a previously committed (rollback) configuration from a device running Junos OS or a device running Junos OS Evolved. To retrieve the configuration using the rollback index, the client application executes the `<get-rollback-information>` RPC with the `<rollback>` element. The `<rollback>` element specifies the rollback index of the previous configuration to retrieve. The value can be from 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49). This operation is equivalent to the `show system rollback` operational mode command.

To request Junos XML-tagged output, which is the default, the application either includes the `<format>xml</format>` element or omits the `<format>` element.

```
<rpc>
    <get-rollback-information>
        <rollback>index-number</rollback>
    </get-rollback-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rollback-information>`, and `<configuration>` elements. The `<ok/>` tag is a side effect of the implementation and does not affect the results.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration attributes>
           <!-- tag elements for complete previous configuration -->
        </configuration>
    </rollback-information>
</rpc-reply>
```

To request formatted ASCII output, the application includes the `<format>text</format>` element.

```
<rpc>
    <get-rollback-information>
        <rollback>index-number</rollback>
        <format>text</format>
    </get-rollback-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rollback-information>`, `<configuration-information>`, and `<configuration-output>` elements.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration-information>
            <configuration-output>
                    /* previous configuration in formatted ASCII*/
            </configuration-output>
        </configuration-information>
    </rollback-information>
</rpc-reply>
```

To request JSON format, the application includes the `<format>json</format>` element.

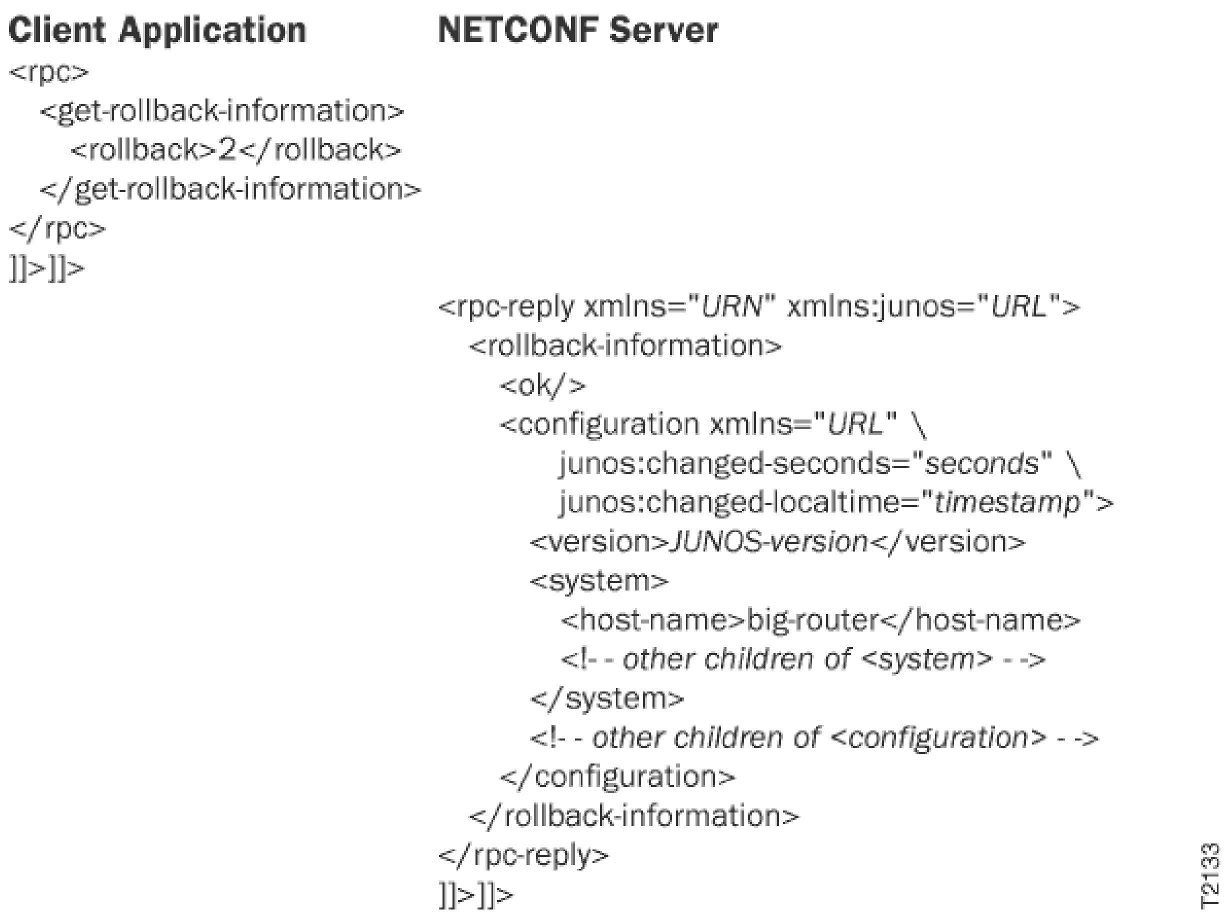
```
<rpc>
    <get-rollback-information>
        <rollback>index-number</rollback>
        <format>json</format>
    </get-rollback-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rollback-information>`, `<configuration-information>`, and `<json-output>` elements.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration-information>
            <json-output>
```

```
                  <!-- JSON data for the complete previous configuration -->
              </json-output>
          </configuration-information>
      </rollback-information>
  </rpc-reply>
```

The following example requests Junos XML-tagged output for the rollback configuration that has an index of 2. In actual output, the *JUNOS-version* variable has a value such as 24.4R1, which is the initial version of Junos OS Release 24.4.



```
Client Application                   NETCONF Server
<rpc>
  <get-rollback-information>
    <rollback>2</rollback>
  </get-rollback-information>
</rpc>
]]>]]>
                                     <rpc-reply xmlns="URN" xmlns:junos="URL">
                                       <rollback-information>
                                         <ok/>
                                         <configuration xmlns="URL" \
                                             junos:changed-seconds="seconds" \
                                             junos:changed-localtime="timestamp">
                                           <version>JUNOS-version</version>
                                           <system>
                                             <host-name>big-router</host-name>
                                             <!- - other children of <system> - ->
                                           </system>
                                           <!- - other children of <configuration> - ->
                                         </configuration>
                                       </rollback-information>
                                     </rpc-reply>
                                     ]]>]]>
```

RELATED DOCUMENTATION

Compare Two Previous (Rollback) Configurations Using NETCONF | 429

Retrieve the Rescue Configuration Using NETCONF | 432

Compare Two Previous (Rollback) Configurations Using NETCONF

In a NETCONF session with a device running Junos OS, to compare the contents of two previously committed (rollback) configurations, a client application emits the Junos XML `<get-rollback-information>` tag element and its child `<rollback>` and `<compare>` tag elements in an `<rpc>` tag element. This operation is equivalent to the `show system rollback` operational mode command with the `compare` option.

The `<rollback>` tag element specifies the index number of the configuration that is the basis for comparison. The `<compare>` tag element specifies the index number of the configuration to compare with the base configuration. Valid values in both tag elements range from 0 (zero, for the most recently committed configuration) through 49:

```
<rpc>
    <get-rollback-information>
        <rollback>index-number</rollback>
        <compare>index-number</compare>
    </get-rollback-information>
</rpc>
]]>]]>
```

NOTE: The output corresponds more logically to the chronological order of changes if the older configuration (the one with the higher index number) is the base configuration. Its index number is enclosed in the `<rollback>` tag element and the index of the more recent configuration is enclosed in the `<compare>` tag element.

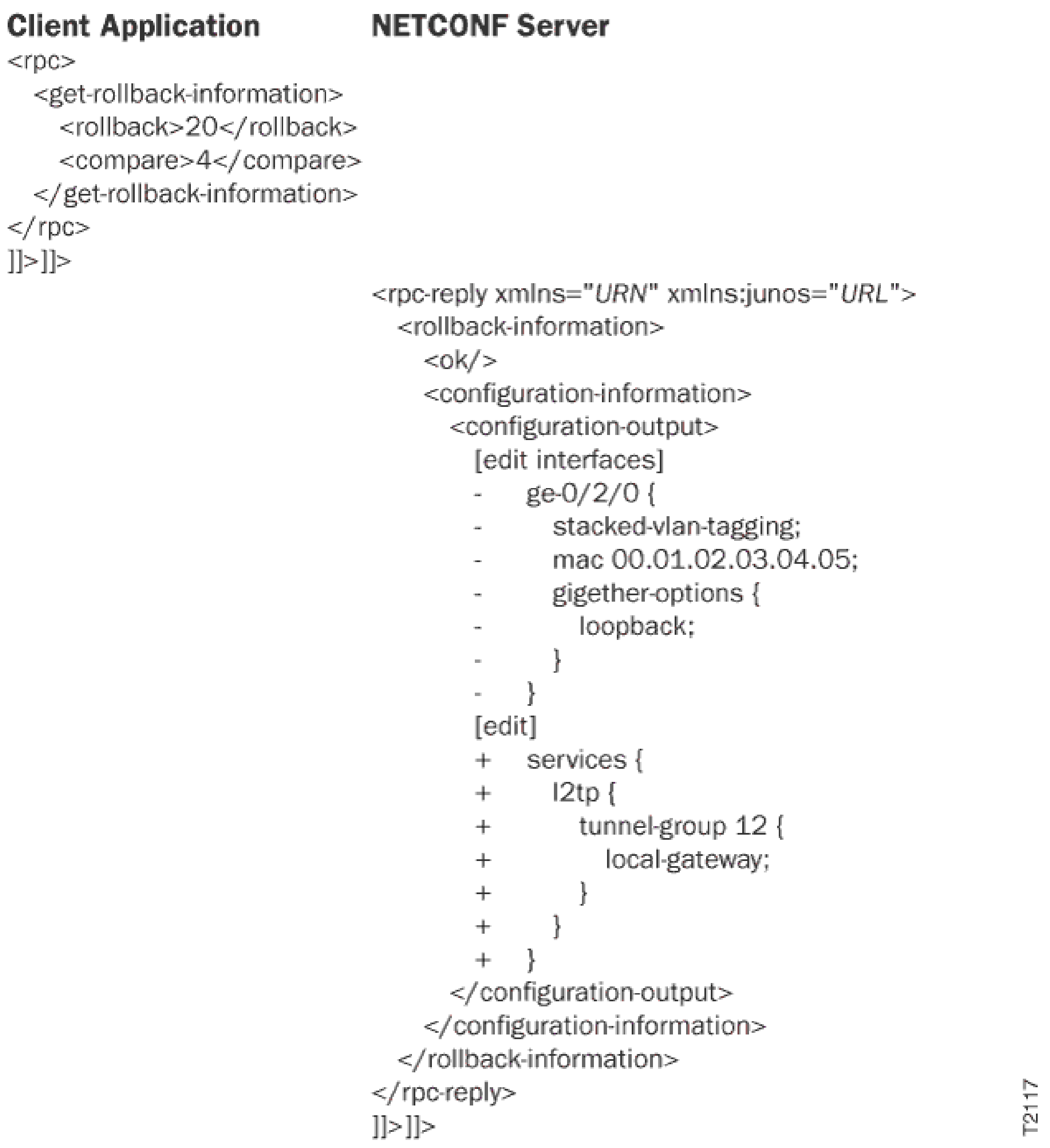
The NETCONF server encloses its response in `<rpc-reply>`, `<rollback-information>`, `<configuration-information>`, and `<configuration-output>` tag elements. The `<ok/>` tag is a side effect of the implementation and does not affect the results.

The information in the `<configuration-output>` tag element is formatted ASCII and includes a banner line (such as `[edit interfaces]`) for each hierarchy level at which the two configurations differ. Each line between banner lines begins with either a plus sign (+) or a minus sign (–). The plus sign indicates that adding the statement to the base configuration results in the second configuration, whereas a minus sign means that removing the statement from the base configuration results in the second configuration.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration-information>
```

```
            <configuration-output>
               /* formatted ASCII representing the changes */
            </configuration-output>
        </configuration-information>
    </rollback-information>
</rpc-reply>
]]>]]>
```

The following example shows how to request a comparison of the rollback configurations that have indexes of 20 and 4.



Client Application

```
<rpc>
   <get-rollback-information>
      <rollback>20</rollback>
      <compare>4</compare>
   </get-rollback-information>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
   <rollback-information>
      <ok/>
      <configuration-information>
         <configuration-output>
            [edit interfaces]
            -     ge-0/2/0 {
            -         stacked-vlan-tagging;
            -         mac 00.01.02.03.04.05;
            -         gigether-options {
            -             loopback;
            -         }
            -     }
            [edit]
            +     services {
            +         l2tp {
            +             tunnel-group 12 {
            +                 local-gateway;
            +             }
            +         }
            +     }
         </configuration-output>
      </configuration-information>
   </rollback-information>
</rpc-reply>
]]>]]>
```

T2117

RELATED DOCUMENTATION

Retrieve a Previous (Rollback) Configuration Using NETCONF | 425

Retrieve the Rescue Configuration Using NETCONF | 432

Retrieve the Rescue Configuration Using NETCONF

SUMMARY

A NETCONF client application can use the `<get-rescue-information>` request tag to retrieve the existing rescue configuration on a Junos device.

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. You use the rescue configuration to revert to a known configuration or as a last resort if the device configuration and the backup configuration files become damaged beyond repair.

You must create a rescue configuration on the device before you can retrieve or use it. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration. You can create a rescue configuration using the following methods:

- In a NETCONF session, use the `<request-save-rescue-configuration>` request tag.
- In the Junos OS CLI, issue the `request system configuration rescue save` operational mode command.

A NETCONF client application can retrieve the rescue configuration from devices running Junos OS or devices running Junos OS Evolved. A client application requests the rescue configuration by emitting an `<rpc>` element and enclosing the Junos XML `<get-rescue-information>` request tag. The operation is equivalent to the `show system configuration rescue` operational mode command.

```
<rpc>
    <get-rescue-information/>
</rpc>
```

By default the server returns the rescue configuration in Junos XML format. To explicitly request XML output, the application can also include the `<format>xml</format>` element.

```
<rpc>
    <get-rescue-information>
        <format>xml</format>
    </get-rescue-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rescue-information>`, and `<configuration>` tag elements. The `<ok/>` tag is a side effect of the implementation and does not affect the results.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rescue-information>
        <ok/>
        <configuration attributes
            <!-- tag elements representing the rescue configuration -->
        </configuration>
    </rescue-information>
</rpc-reply>
```

To request the rescue configuration in formatted ASCII output, the application includes the `<format>` element with the value `text`.

```
<rpc>
    <get-rescue-information>
        <format>text</format>
    </get-rescue-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rescue-information>`, `<configuration-information>`, and `<configuration-output>` tag elements.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rescue-information>
        <ok/>
        <configuration-information>
            <configuration-output>
                    /* formatted ASCII for the rescue configuration*/
            </configuration-output>
        </configuration-information>
    </rescue-information>
</rpc-reply>
```

To request the rescue configuration in JSON format, the application includes the `<format>` element with the value `json`.

```
<rpc>
    <get-rescue-information>
```

```
        <format>json</format>
    </get-rescue-information>
</rpc>
```

The NETCONF server encloses its response in `<rpc-reply>`, `<rescue-information>`, `<configuration-information>`, and `<json-output>` elements.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rescue-information>
        <ok/>
        <configuration-information>
            <json-output>
                {
                    "configuration" : {
                        <!-- JSON data representing the rescue configuration -->
                    }
                }
            </json-output>
        </configuration-information>
    </rescue-information>
</rpc-reply>
```

RELATED DOCUMENTATION

Retrieve a Previous (Rollback) Configuration Using NETCONF | 425

Compare Two Previous (Rollback) Configurations Using NETCONF | 429

Request an XML Schema for the Configuration Hierarchy Using NETCONF

IN THIS SECTION

The Junos configuration schema represents all configuration elements available in the version of the OS that is running on a device. To determine the Junos OS or Junos OS Evolved version, emit the `<get-software-information>` operational request tag. Client applications can use the schema simply to learn which configuration statements are available in their version of Junos OS or Junos OS Evolved. Client applications can also use the schema to validate the configuration on a device.

The schema does not indicate which elements are actually configured. Moreover, the schema does not indicate that you can even configure an element on that type of device (some configuration statements are available only on certain device types). To request the set of currently configured elements and their settings, emit the `<get-config>` tag element instead.

Explaining the structure and notational conventions of the XML Schema language is beyond the scope of this document. For information, see *XML Schema Part 0: Primer*. The primer provides a basic introduction and lists the formal specifications where you can find detailed information.

Request an XML Schema for the Configuration Hierarchy

A NETCONF client application can request an XML Schema-language representation of the entire configuration hierarchy on a device running Junos OS or a device running Junos OS Evolved. To request the XML schema, a client application emits an `<rpc>` element and encloses the Junos XML `<get-xnm-information>` element. The `<get-xnm-information>` element encloses the `<type>` and `<namespace>` child elements with the indicated values.

```
<rpc>
    <get-xnm-information>
        <type>xml-schema</type>
        <namespace>junos-configuration</namespace>
    </get-xnm-information>
</rpc>
```

The NETCONF server encloses the XML schema in `<rpc-reply>` and `<xsd:schema>` tags.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <xsd:schema>
        <!-- tag elements for the Junos XML schema -->
    </xsd:schema>
</rpc-reply>
```

Create the junos.xsd File

Most of the tag elements defined in the schema returned in the `<xsd:schema>` tag belong to the default namespace for Junos OS configuration elements. However, at least one tag, `<junos:comment>`, belongs to a different namespace: `http://xml.juniper.net/junos/`*`Junos-version`*`/junos`. By XML convention, a schema describes only one namespace, so schema validators need to import information about any additional namespaces before they can process the schema.

The `<xsd:schema>` element encloses the `<xsd:import>` tag, which references the **junos.xsd** file. This file contains the required information about the `junos` namespace. For example, the following `<xsd:import>` tag specifies the file for Junos OS Release 20.4R1:

```
<xsd:import schemaLocation="junos.xsd" namespace="http://xml.juniper.net/junos/20.4R1/junos"/>
```

To enable the schema validator to interpret the `<xsd:import>` tag, the **junos.xsd** file must exist. You must manually create a file called **junos.xsd** in the directory where the Junos configuration schema resides. Include the following text in the file. Do not use line breaks in the list of attributes in the opening `<xsd:schema>` tag. Line breaks appear in the following example for legibility only. For the *Junos-version* variable, substitute the release number of the Junos OS or Junos OS Evolved release running on the device (for example, 20.4R1).

```
<?xml version="1.0" encoding="us-ascii"?>
<xsd:schema elementFormDefault="qualified" \
        attributeFormDefault="unqualified" \
        xmlns:xsd="http://www.w3.org/2001/XMLSchema" \
        targetNamespace="http://xml.juniper.net/junos/Junos-version/junos">
    <xsd:element name="comment" type="xsd:string"/>
</xsd:schema>
```

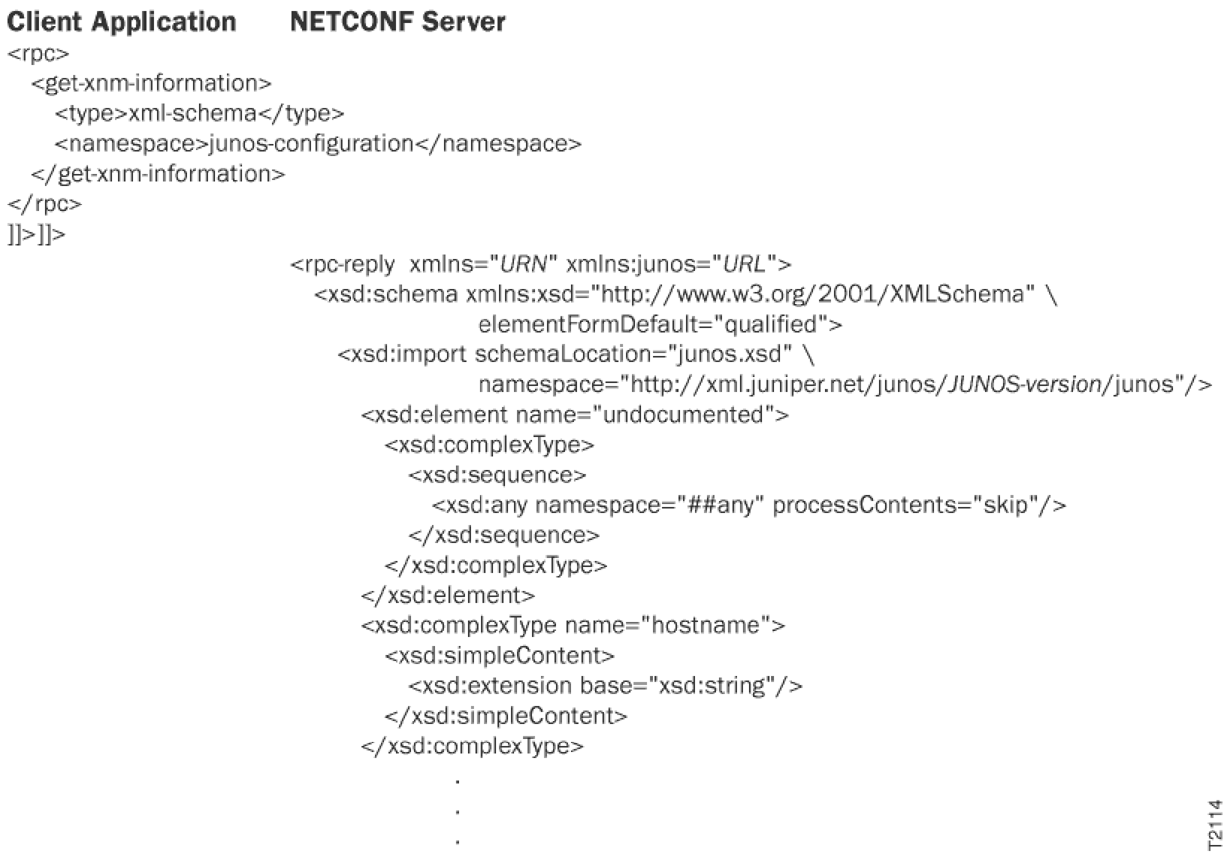
NOTE: Schema validators might not be able to process the schema if they cannot locate or open the **junos.xsd** file.

Whenever you change the version of Junos OS running on the device, remember to update the *Junos-version* variable in the **junos.xsd** file to match.

Example: Request an XML Schema

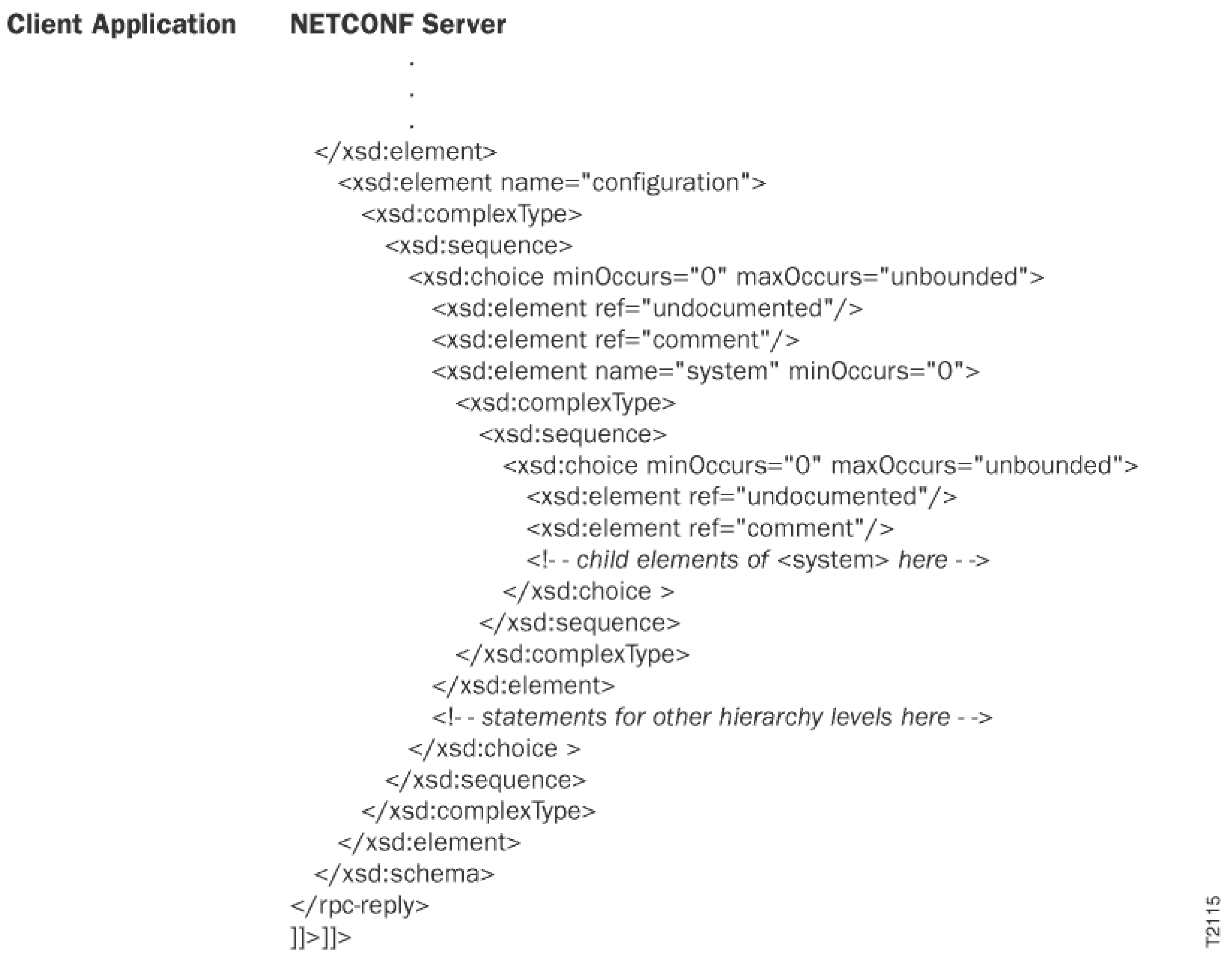
The following examples request the Junos OS configuration schema. In the NETCONF server's response, the first `<xsd:element>` statement defines the `<undocumented>` Junos XML tag element. This element can be enclosed in most other container tag elements defined in the schema (container tag elements are defined as `<xsd:complexType>`).

The attributes in the opening tags of the server's response appear on multiple lines for legibility only. Also, in actual output the *JUNOS-version* variable is replaced by a value such as 20.4R1 for the initial version of Junos OS Release 20.4.



```
<rpc>
  <get-xnm-information>
    <type>xml-schema</type>
    <namespace>junos-configuration</namespace>
  </get-xnm-information>
</rpc>
]]>]]>
                        <rpc-reply  xmlns="URN" xmlns:junos="URL">
                          <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" \
                                        elementFormDefault="qualified">
                            <xsd:import schemaLocation="junos.xsd" \
                                        namespace="http://xml.juniper.net/junos/JUNOS-version/junos"/>
                              <xsd:element name="undocumented">
                                <xsd:complexType>
                                  <xsd:sequence>
                                    <xsd:any namespace="##any" processContents="skip"/>
                                  </xsd:sequence>
                                </xsd:complexType>
                              </xsd:element>
                              <xsd:complexType name="hostname">
                                <xsd:simpleContent>
                                  <xsd:extension base="xsd:string"/>
                                </xsd:simpleContent>
                              </xsd:complexType>
                                      .
                                      .
                                      .
```

Another `<xsd:element>` statement near the beginning of the schema defines the Junos XML `<configuration>` element. It encloses the `<xsd:element>` statement that defines the `<system>` element, which corresponds to the `[edit system]` hierarchy level. For brevity, the output omits the statements corresponding to other hierarchy levels.


Client Application
NETCONF Server
T2115

```
        .
        .
        .
  </xsd:element>
    <xsd:element name="configuration">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="undocumented"/>
            <xsd:element ref="comment"/>
            <xsd:element name="system" minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:choice minOccurs="0" maxOccurs="unbounded">
                    <xsd:element ref="undocumented"/>
                    <xsd:element ref="comment"/>
                    <!- - child elements of <system> here - ->
                  </xsd:choice >
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <!- - statements for other hierarchy levels here - ->
          </xsd:choice >
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</rpc-reply>
]]>]]>
```

RELATED DOCUMENTATION

Request Configuration Data Using NETCONF | 400

5

PART

NETCONF Utilities

CHAPTER 15

NETCONF Perl Client

IN THIS CHAPTER

Understanding the NETCONF Perl Client and Sample Scripts

SUMMARY

Administrators familiar with Perl can use the NETCONF Perl client API to create Perl applications that manage Junos devices using NETCONF.

IN THIS SECTION

Devices running Junos OS and devices running Junos OS Evolved support the NETCONF XML management protocol. The NETCONF protocol enables client applications to request information and change the configuration on network devices. The protocol uses an Extensible Markup Language (XML)-based data encoding for the configuration data and remote procedure calls (RPCs). The Juniper Networks NETCONF Perl API enables programmers familiar with the Perl programming language to create their own Perl applications to manage Junos devices using NETCONF.

The NETCONF Perl client is hosted on GitHub and CPAN. It is release-independent, and it can manage devices running any version of Junos OS or Junos Evolved. The following sections discuss the NETCONF Perl client modules and sample scripts.

NETCONF Perl Client Modules

Table 21 on page 441 summarizes the modules in the NETCONF Perl library. The `Net::Netconf::Manager` module provides an object-oriented interface for communicating with the NETCONF server on Junos devices. The module enables you to easily connect to the device, establish a NETCONF session, and execute operational and configuration requests. Client applications only directly invoke the

Net::Netconf::Manager object. When the client application creates a Manager object, it supplies the device name and the login name to use when accessing the device. The login name determines the client application's access level on the device.

Table 21: NETCONF Perl Modules

Module	Description
Access	Creates an Access object based on the access method type specified when instantiating the object. The module is responsible for calling the connect() method to establish a session with the NETCONF server at the destination host and for exchanging hello packets with the server after the session is established.
Constants	Declares all NETCONF constants.
Device	Implements an object-oriented interface to the NETCONF API supported by devices running Junos OS and devices running Junos OS Evolved. Objects of this class represent the local side of the connection to the device, which communicates to the client using the NETCONF protocol.
EzEditXML	Facilitates the development of XML documents for both operational and configuration requests. The module uses XML::LibXML as a base library, but provides Junos OS CLI-specific features to manipulate the configuration, corresponding to the CLI commands: delete, activate, deactivate, insert, and rename.
Manager	Instantiates and returns a NETCONF or Junos XML Device object depending on which server is requested.
SAXHandler	SAX-based parser that parses responses from the NETCONF server.
SSH	Provides SSH access to a Net::Netconf::Access instance, and manages the SSH connection with the destination host. The underlying mechanism for managing the SSH connection is based on OpenSSH.
Trace	Provides tracing levels and enables tracing based on the requested debug level.

Client applications can also leverage Perl modules in the public domain to ease the development of NETCONF Perl client applications. Because NETCONF uses XML-based data encoding, client applications can use the many Perl modules that manipulate XML data.

You can use the NETCONF Perl client to create Perl applications that connect to a device, establish a NETCONF session, and execute operations. The communication between the client and the NETCONF server on the device through the NETCONF Perl API involves the following steps:

- Establishing a NETCONF session over SSHv2 between the client application and the NETCONF server on the Junos device.
- Creating RPCs corresponding to requests and sending the requests to the NETCONF server.
- Receiving and processing the RPC replies from the NETCONF server.

Sample Scripts

The NETCONF Perl distribution includes an **examples** directory with the following sample scripts that illustrate how to use the modules to perform various functions. For instructions on running the scripts, see the **README** file in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

- **diagnose_bgp/diagnose_bgp.pl**—Illustrates how to monitor the status of the device and diagnose problems. The script extracts and displays information about a device's unestablished BGP peers from the full set of BGP configuration data.
- **get_chassis_inventory/get_chassis_inventory.pl**—Illustrates how to use a predefined query to request information from a device. The sample script invokes the `get_chassis_inventory` query with the `detail` option. The query requests the same information as returned by the Junos XML `<get-chassis-inventory><detail/></get-chassis-inventory>` request or the CLI operational mode command `show chassis hardware detail`.
- **edit_configuration/edit_configuration.pl**—Illustrates how to configure the device by loading a file that contains Junos XML-formatted configuration data. The distribution includes a sample configuration file, **config.xml**. However, you can specify a different configuration file on the command line when you invoke the script.

RELATED DOCUMENTATION

Install the NETCONF Perl Client | 443

Write NETCONF Perl Client Applications | 444

Install the NETCONF Perl Client

The Juniper Networks NETCONF Perl API enables programmers familiar with the Perl programming language to create their own Perl applications to manage and configure Junos devices. The NETCONF Perl client, which is available on GitHub and through the Comprehensive Perl Archive Network (CPAN), is independent of the Junos OS release running on the managed devices. You can use the same client installation to manage devices running any Junos OS release.

The NETCONF Perl distribution uses the same directory structure for Perl modules as CPAN. The distribution includes a **lib** directory for the `NET::Netconf` module and its supporting files, and an **examples** directory for sample scripts. You install the NETCONF Perl distribution on a device running a Unix-like OS. After you install the software, you can create Perl applications to connect to a device running Junos OS, establish a NETCONF session, and execute operations.

For information about installing the NETCONF Perl API, follow the instructions in the README file located in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

RELATED DOCUMENTATION

Understanding the NETCONF Perl Client and Sample Scripts | **440**

CHAPTER 16

Develop NETCONF Perl Client Applications

IN THIS CHAPTER

Write NETCONF Perl Client Applications

The Juniper Networks NETCONF Perl client enables programmers familiar with the Perl programming language to create their own Perl applications to manage and configure Junos devices. The `Net::Netconf::Manager` module provides a release-independent object-oriented interface for communicating with a NETCONF server on managed devices. The module enables you to connect to a device, establish a NETCONF session, and execute operational and configuration requests.

The following outline lists the basic tasks involved in writing a NETCONF Perl client application that manages Junos devices. Each task provides a link to more detailed information about performing that task.

1. Import Perl Modules and Declare Constants—"Import Perl Modules and Declare Constants in NETCONF Perl Client Applications" on page 446
2. Connect to the NETCONF Server—"Connect to the NETCONF Server in Perl Client Applications" on page 447 and "Collect Parameters Interactively in NETCONF Perl Client Applications" on page 450
3. Submit Requests to the NETCONF Server—"Submit a Request to the NETCONF Server in Perl Client Applications" on page 454

4. Parse and Format the Response from the NETCONF Server—"Parse the NETCONF Server Response in Perl Client Applications" on page 466
5. Close the Connection to the NETCONF Server—"Close the Connection to the NETCONF Server in Perl Client Applications" on page 467

The tasks are illustrated in the following example, which uses the `Net::Netconf::Manager` object to request information from a device running Junos OS. The example presents the minimum code required to execute a simple query.

1. Import required modules and declare constants.

```
use strict;
use Carp;
use Net::Netconf::Manager;
```

2. Create a `Manager` object and connect to the device.

```
my %deviceinfo = (
   access => "ssh",
   login => "johndoe",
   password => "password123",
   hostname => "Router1"
);
my $jnx = new Net::Netconf::Manager(%deviceinfo);

unless ( ref $jnx ) {
   croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}
```

3. Construct the query and send it to the NETCONF server.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();
```

4. Process the response as needed.

```
print "Server response: \n $jnx->{'server_response'} \n";
```

5. Disconnect from the NETCONF server.

   ```
   $jnx->disconnect();
   ```

RELATED DOCUMENTATION

Understanding the NETCONF Perl Client and Sample Scripts | **440**

Import Perl Modules and Declare Constants in NETCONF Perl Client Applications

When creating a NETCONF Perl client application, include the following statement at the start of the application. This statement imports the functions provided by the `Net::Netconf::Manager` object, which the application uses to connect to the NETCONF server on a device.

```
use Net::Netconf::Manager;
```

Include statements to import other Perl modules as appropriate for your application. For example, several of the sample scripts included in the NETCONF Perl distribution import the following standard Perl modules, which include functions that handle input from the command line:

- Carp—Includes functions for user error warnings.
- Getopt::Std—Includes functions for reading in keyed options from the command line.
- Term::ReadKey—Includes functions for controlling terminal modes, for example suppressing onscreen echo of a typed string such as a password.

If the application uses constants, declare their values at this point. For example, the sample script **diagnose_bgp.pl** includes the following statement to declare a constant for the access method:

```
use constant VALID_ACCESS_METHOD => 'ssh';
```

The **edit_configuration.pl** sample script includes the following statements to declare constants for reporting return codes and the status of the configuration database:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Connect to the NETCONF Server in Perl Client Applications | 447

Connect to the NETCONF Server in Perl Client Applications

IN THIS SECTION

The following sections explain how to use the `NET::Netconf::Manager` object in a Perl client application to connect to the NETCONF server on a device running Junos OS:

Satisfy Protocol Prerequisites

The NETCONF server supports several access protocols. For each connection to the NETCONF server on a device running Junos OS, the application must specify the protocol it is using. Perl client applications can communicate with the NETCONF server via SSH only.

Before your application can run, you must satisfy the prerequisites for SSH. This involves enabling NETCONF on the device by configuring the `set system services netconf ssh` statement.

Group Requests

Establishing a connection to the NETCONF server on a device running Junos OS is one of the more time-intensive and resource-intensive functions performed by an application. If the application sends multiple requests to a device, it makes sense to send all of them within the context of one connection. If your application sends the same requests to multiple devices, you can structure the script to iterate through either the set of devices or the set of requests. Keep in mind, however, that your application can effectively send only one request to one NETCONF server at a time. This is because the `NET::Netconf::Manager` object does not return control to the application until it receives the closing `</rpc-reply>` tag that represents the end of the NETCONF server's response to the current request.

Obtain and Record Parameters Required by the NET::Netconf::Manager Object

The `NET::Netconf::Manager` object takes the following required parameters, specified as keys in a Perl hash:

- `access`—The access protocol to use when communicating with the NETCONF server. Before the application runs, satisfy the SSH prerequisites.
- `hostname`—The name of the device to which to connect. For best results, specify either a fully-qualified hostname or an IP address.
- `login`—The username under which to establish the connection to the NETCONF server and issue requests. The username must already exist on the specified device and have the permission bits necessary for making the requests invoked by the application.
- `password`—The password corresponding to the username.

The sample scripts in the NETCONF Perl distribution record the parameters in a Perl hash called `%deviceinfo`, declared as follows:

```
my %deviceinfo = (
        'access' => $access,
        'login' => $login,
        'password' => $password,
        'hostname' => $hostname,
);
```

The sample scripts included in the NETCONF Perl client distribution obtain the parameters from options entered on the command line by a user. For more information about collecting parameter values interactively, see "Collect Parameters Interactively in NETCONF Perl Client Applications" on page 450.

Your application can also obtain values for the parameters from a file or database, or you can hardcode one or more of the parameters into the application code if they are constant.

Obtaining Application-Specific Parameters

In addition to the parameters required by the `NET::Netconf::Manager` object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the NETCONF server in response to a request.

As with the parameters required by the `NET::Netconf::Manager` object, the client application can hardcode the values in the application code, obtain them from a file, or obtain them interactively. The sample scripts obtain values for these parameters from command-line options in the same manner as they obtain the parameters required by the `NET::Netconf::Manager` object. Several examples follow.

The following line enables a debugging trace if the user includes the `-d` command-line option:

```
my $debug_level = $opt{'d'};
```

The following line sets the `$outputfile` variable to the value specified by the `-o` command-line option. It names the local file to which the NETCONF server's response is written. If the `-o` option is not provided, the variable is set to the empty string.

```
my $outputfile = $opt{'o'} || "";
```

Establishing the Connection

After obtaining values for the parameters required for the `NET::Netconf::Manager` object, each sample script records them in the `%deviceinfo` hash.

```
my %deviceinfo = (
      'access' => $access,
      'login' => $login,
      'password' => $password,
      'hostname' => $hostname,
);
```

The script then invokes the NETCONF-specific `new` subroutine to create a `NET::Netconf::Manager` object and establish a connection to the specified routing, switching, or security platform. If the connection attempt fails (as tested by the `ref` operator), the script exits.

```
my $jnx = new Net::Netconf::Manager(%deviceinfo);
unless (ref $jnx) {
    croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Import Perl Modules and Declare Constants in NETCONF Perl Client Applications | 446

Collect Parameters Interactively in NETCONF Perl Client Applications | 450

Submit a Request to the NETCONF Server in Perl Client Applications | 454

Close the Connection to the NETCONF Server in Perl Client Applications | 467

Collect Parameters Interactively in NETCONF Perl Client Applications

In a NETCONF Perl client application, a script can interactively obtain the parameters required by the `NET::Netconf::Manager` object from the command-line.

The NETCONF Perl distribution includes several sample Perl scripts to perform various functions on Junos devices. Each sample script obtains the parameters required by the `NET::Netconf::Manager` object from command-line options provided by the user who invokes the script. The scripts use the `getopts` function defined in the `Getopt::Std` Perl module to read the options from the command line and then record the options in a Perl hash called `%opt`. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

The following example references the **get_chassis_inventory.pl** sample script from the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl/tree/master/examples/get_chassis_inventory.

The first parameter to the `getopts` function defines the acceptable options, which vary depending on the application. A colon after the option letter indicates that it takes an argument.

The second parameter, `\%opt`, specifies that the values are recorded in the `%opt` hash. If the user does not provide at least one option, provides an invalid option, or provides the `-h` option, the script invokes the `output_usage` subroutine, which prints a usage message to the screen.

```
my %opt;
getopts('l:p:d:f:m:o:h', \%opt) || output_usage();
output_usage() if $opt{'h'};
```

The following code defines the `output_usage` subroutine for the **get_chassis_inventory.pl** sample script. The contents of the `my $usage` definition and the `Where` and `Options` sections are specific to the script, and differ for each application.

```
sub output_usage
{
    my $usage = "Usage: $0 [options] <target>

Where:

  <target>   The hostname of the target device.

Options:

  -l <login>    A login name accepted by the target device.
  -p <password> The password for the login name.
  -m <access>   Access method. The only supported method is 'ssh'.
  -f <xmlfile>  The name of the XML file to print server response to.
                Default: chassis_inventory.xml
  -o <filename> output is written to this file instead of standard output.
  -d <level>    Debug level [1-6]\n\n";

    croak $usage;
}
```

The **get_chassis_inventory.pl** script includes the following code to obtain values from the command line for the parameters required by the `NET::Netconf::Manager` object. A detailed discussion of the various functional units follows the complete code sample.

```
# Get the hostname
my $hostname = shift || output_usage();
```

```
# Get the access method, can be ssh only
my $access = $opt{'m'} || 'ssh';
use constant VALID_ACCESS_METHOD => 'ssh';
output_usage() unless (VALID_ACCESS_METHOD =~ /$access/);

# Check for login name. If not provided, prompt for it
my $login = "";
if ($opt{'l'}) {
    $login = $opt{'l'};
} else {
    print STDERR "login: ";
    $login = ReadLine 0;
    chomp $login;
}

# Check for password. If not provided, prompt for it
my $password = "";
if ($opt{'p'}) {
    $password = $opt{'p'};
} else {
    print STDERR "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print STDERR "\n";
}
```

In the first line of the preceding code sample, the script uses the Perl `shift` function to read the hostname from the end of the command line. If the hostname is missing, the script invokes the `output_usage` subroutine to print the usage message, which specifies that a hostname is required.

```
my $hostname = shift || output_usage();
```

The script next determines which access protocol to use, setting the `$access` variable to the value of the `-m` command-line option. If the specified value does not match the only valid value defined by the `VALID_ACCESSES` constant, the script invokes the `output_usage` subroutine to print the usage message.

```
my $access = $opt{'m'} || 'ssh';
use constant VALID_ACCESS_METHOD => 'ssh';
output_usage() unless (VALID_ACCESS_METHOD =~ /$access/);
```

The script then determines the username, setting the `$login` variable to the value of the `-l` command-line option. If the option is not provided, the script prompts for it and uses the `ReadLine` function (defined in the standard Perl `Term::ReadKey` module) to read it from the command line.

```
my $login = "";
if ($opt{'l'}) {
    $login = $opt{'l'};
} else {
    print STDERR "login: ";
    $login = ReadLine 0;
    chomp $login;
}
```

The script finally determines the password for the username, setting the `$password` variable to the value of the `-p` command-line option. If the option is not provided, the script prompts for it. It uses the `ReadMode` function (defined in the standard Perl `Term::ReadKey` module) twice: first to prevent the password from echoing visibly on the screen, and then to return the shell to normal (echo) mode after it reads the password.

```
my $password = "";
if ($opt{'p'}) {
    $password = $opt{'p'};
} else {
    print STDERR "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print STDERR "\n";
}
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Submit a Request to the NETCONF Server in Perl Client Applications

IN THIS SECTION

In a NETCONF Perl client application, after establishing a connection to the NETCONF server, the client application can execute operational or configuration commands on a device running Junos OS to request operational information or change the configuration. The NETCONF Perl API supports a set of methods that correspond to CLI operational mode commands and NETCONF configuration operations. To execute a command, the client application invokes the Perl method corresponding to that command.

See the following sections for more information:

Mapping Junos OS Commands and NETCONF Operations to Perl Methods

Most operational commands have a corresponding Junos XML request tag. You can find the Junos XML request tags for operational commands by using the Junos XML API Explorer. You can also display the Junos XML request tags in the CLI. Once you obtain the request tag, you can map it to the corresponding Perl method name.

To display the Junos XML request tags for a command in the CLI, issue the command and include the `| display xml rpc` option. The following example displays the request tag for the `show route` command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
</rpc-reply>
```

You can map the request tag for an operational command to a Perl method name. To derive the method name, replace any hyphens in the request tag with underscores, and remove the enclosing angle brackets. For example, the `<get-route-information>` request tag maps to the `get_route_information` method name.

Similarly, NETCONF protocol operations map to Perl method names in the same manner. For example, the `<edit-config>` operation maps to the `edit_config` method name.

Providing Method Options

Perl methods can have one or more options. The following section describes the notation that an application uses to define a method's options in a NETCONF Perl client application.

- A method without options is defined as `$NO_ARGS`, as in the following entry for the `get_autoinstallation_status_information` method:

```
## Method : get_autoinstallation_status_information
## Returns: <autoinstallation-status-information>
## Command: "show system autoinstallation status"
get_autoinstallation_status_information => $NO_ARGS,
```

  To invoke a method without options, the client application follows the method name with an empty set of parentheses, as in the following example:

```
$jnx->get_autoinstallation_status_information();
```

- A fixed-form option is defined as type `$TOGGLE`. In the following example, the `get_ancp_neighbor_information` method has two fixed-form options, `brief` and `detail`:

```
## Method : get_ancp_neighbor_information
## Returns: <ancp-neighbor-information>
## Command: "show ancp neighbor"
get_ancp_neighbor_information => {
    brief => $TOGGLE,
    detail => $TOGGLE,
}
```

To include a fixed-form option when invoking a method, set the option equal to the string `'True'`, as in the following example:

```
$jnx->get_ancp_neighbor_information(brief => 'True');
```

NOTE: When using the release-dependent NETCONF Perl distribution, to include a fixed-form option when invoking a method, set the option equal to the value 1 (one).

- An option with a variable value is defined as type `$STRING`. In the following example, the `get_cos_drop_profile_information` method takes the `profile_name` argument:

```
## Method : get_cos_drop_profile_information
## Returns: <cos-drop-profile-information>
## Command: "show class-of-service drop-profile"
get_cos_drop_profile_information => {
    profile_name => $STRING,
},
```

To include a variable value when invoking a method, enclose the value in single quotes, as in the following example:

```
$jnx->get_cos_drop_profile_information(profile_name => 'user-drop-profile');
```

- A set of configuration statements or corresponding tag elements is defined as type `$DOM`. In the following example, the `get_config` method takes a set of configuration statements (along with two attributes):

```
'get_config' => {
     'source' => $DOM_STRING,
     'source_url' => $URL_STRING,
     'filter' => $DOM
},
```

A DOM object is XML code:

```
my $xml_string = "
<filter type=\"subtree\">
```

```
<configuration>
  <protocols>
    <bgp></bgp>
  </protocols>
</configuration>
</filter>
";

my %queryargs = (
   'source' => "running",
   'filter' => $xml_string,
);
```

This generates the following RPC request:

```
<rpc message-id='1'>
<get-config>
   <source> <running/> </source>
   <filter type="subtree">
      <configuration>
         <protocols>
            <bgp></bgp>
         </protocols>
      </configuration>
   </filter>
</get-config>
</rpc>
```

A method can have a combination of fixed-form options, options with variable values, and a set of configuration statements. For example, the `get_forwarding_table_information` method has four fixed-form options and five options with variable values:

```
## Method : get_forwarding_table_information
## Returns: <forwarding-table-information>
## Command: "show route forwarding-table"
get_forwarding_table_information => {
    detail => $TOGGLE,
    extensive => $TOGGLE,
    multicast => $TOGGLE,
    family => $STRING,
    vpn => $STRING,
    summary => $TOGGLE,
    matching => $STRING,
    destination => $STRING,
    label => $STRING,
},
```

Submitting a Request

The following code illustrates the recommended way to send a configuration request to the NETCONF server and shows how to handle error conditions. The `$jnx` variable is defined to be a `NET::Netconf::Manager` object. The sample code, which is taken from the **edit_configuration.pl** sample script, locks the candidate configuration, loads the configuration changes, commits the changes, and then unlocks the configuration database and disconnects from the NETCONF server. You can view the complete **edit_configuration.pl** script in the **examples/edit_configuration** directory in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

```
my $res; # Netconf server response

# connect to the Netconf server
my $jnx = new Net::Netconf::Manager(%deviceinfo);
unless (ref $jnx) {
    croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}

# Lock the configuration database before making any changes
print "Locking configuration database ...\n";
my %queryargs = ( 'target' => 'candidate' );
$res = $jnx->lock_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    graceful_shutdown($jnx, STATE_CONNECTED, REPORT_FAILURE);
}

# Load the configuration from the given XML file
print "Loading configuration from $xmlfile \n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
}
```

```
# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";

%queryargs = (
        'target' => 'candidate'
     );

# If we are in text mode, use config-text arg with wrapped
# configuration-text, otherwise use config arg with raw
# XML
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration-text>' . $config
    . '</configuration-text>';
} else {
  $queryargs{'config'} = $config;
}

$res = $jnx->edit_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    # Get the error
    my $error = $jnx->get_first_error();
    get_error_info(%$error);
    # Disconnect
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
}

# Commit the changes
print "Committing the <edit-config> changes ...\n";
$jnx->commit();
if ($jnx->has_error) {
    print "ERROR: Failed to commit the configuration.\n";
    graceful_shutdown($jnx, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

# Unlock the configuration database and
# disconnect from the Netconf server
print "Disconnecting from the Netconf server ...\n";
graceful_shutdown($jnx, STATE_LOCKED, REPORT_SUCCESS);
```

RELATED DOCUMENTATION

Example: Request an Inventory of Hardware Components Using a NETCONF Perl Client Application

The NETCONF Perl distribution includes several sample Perl scripts to perform various functions on devices running Junos OS. The **get_chassis_inventory.pl** script retrieves and displays a detailed inventory of the hardware components installed in a routing, switching, or security platform. It is equivalent to issuing the `show chassis hardware detail` operational mode command in the Junos OS CLI. This topic describes the portion of the script that executes the query.

After establishing a connection to the NETCONF server, the script sends the `get_chassis_inventory` request and includes the `detail` argument.

```
my $query = "get_chassis_inventory";
my %queryargs = ( 'detail' => 'True' );
```

NOTE: When using the release-independent NETCONF Perl distribution, to include a fixed-form option when invoking a method, set the option equal to the value 1 (one).

The script sends the query and assigns the return value to the `$res` variable. The script first prints the RPC request and response to standard output, then it prints the response to the specified file. The script then checks for and prints any error encountered.

```
my $res; # Netconf server response

# send the command and get the server response
my $res = $jnx->$query(%queryargs);
print "Server request: \n $jnx->{'request'}\n Server response: \n $jnx->{'server_response'} \n";

# print the server response into xmlfile
print_response($xmlfile, $jnx->{'server_response'});
```

```
# See if you got an error
if ($jnx->has_error) {
    croak "ERROR: in processing request \n $jnx->{'request'} \n";
} else {
    print "Server Response:";
    print "$res";
}

# Disconnect from the Netconf server
$jnx->disconnect();
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Submit a Request to the NETCONF Server in Perl Client Applications | 454

Example: Change the Configuration Using a NETCONF Perl Client Application

IN THIS SECTION

The NETCONF Perl distribution includes several sample Perl scripts to perform various functions on devices running Junos OS. The **edit_configuration.pl** script locks, modifies, uploads, and commits the configuration on a device. It uses the basic structure for sending requests but also defines a `graceful_shutdown` subroutine that handles errors. The following sections describe the different functions that the script performs:

Handling Error Conditions

The `graceful_shutdown` subroutine in the **edit_configuration.pl** script handles errors encountered in the NETCONF session. It employs the following additional constants:

```
# query execution status constants
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

The first two `if` statements in the subroutine refer to the `STATE_CONFIG_LOADED` and `STATE_LOCKED` conditions, which apply specifically to loading a configuration in the **edit_configuration.pl** script.

```
sub graceful_shutdown
{
   my ($jnx, $state, $success) = @_;
   if ($state >= STATE_CONFIG_LOADED) {
       # We have already done an <edit-config> operation
       # - Discard the changes
       print "Discarding the changes made ...\n";
       $jnx->discard_changes();
       if ($jnx->has_error) {
           print "Unable to discard <edit-config> changes\n";
       }
   }

   if ($state >= STATE_LOCKED) {
       # Unlock the configuration database
       $jnx->unlock_config();
       if ($jnx->has_error) {
           print "Unable to unlock the candidate configuration\n";
       }
   }

   if ($state >= STATE_CONNECTED) {
       # Disconnect from the Netconf server
       $jnx->disconnect();
   }

   if ($success) {
       print "REQUEST succeeded !!\n";
   } else {
       print "REQUEST failed !!\n";
   }

   exit;
}
```

Locking the Configuration

The main section of the **edit_configuration.pl** script begins by establishing a connection to a NETCONF server. It then invokes the `lock_configuration` method to lock the configuration database. If an error occurs, the script invokes the `graceful_shutdown` subroutine described in "Handling Error Conditions " on page 462.

```
print "Locking configuration database ...\n";
my %queryargs = ( 'target' => 'candidate' );
$res = $jnx->lock_config(%queryargs);
# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    graceful_shutdown($jnx, STATE_CONNECTED, REPORT_FAILURE);
}
```

Reading In the Configuration Data

In the following code sample, the **edit_configuration.pl** script reads in and parses a file that contains Junos XML configuration tag elements or ASCII-formatted statements. A detailed discussion of the functional subsections follows the complete code sample.

```
# Load the configuration from the given XML file
print "Loading configuration from $xmlfile \n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
}

# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";
```

```
%queryargs = (
                 'target' => 'candidate'
             );

# If we are in text mode, use config-text arg with wrapped
# configuration-text, otherwise use config arg with raw XML
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration text> . $config . </configuration-text>';
} else {
  $queryargs{'config'} = $config;
```

The first subsection of the preceding code sample verifies the existence of the file containing configuration data. The name of the file was previously obtained from the command line and assigned to the `$xmlfile` variable. If the file does not exist, the script invokes the `graceful_shutdown` subroutine.

```
print "Loading configuration from $xmlfile \n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
}
```

The script then invokes the `read_xml_file` subroutine, which opens the file for reading and assigns its contents to the `$config` variable. The `queryargs` key `target` is set to the value `candidate`. When the script calls the `edit_configuration` method, the candidate configuration is edited.

```
# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";

%queryargs = (
                 'target' => 'candidate'
             );
```

If the `-t` command-line option was included when the **edit_configuration.pl** script was invoked, the file referenced by the `$xmlfile` variable should contain ASCII-formatted configuration statements like those returned by the CLI configuration-mode `show` command. If the configuration statements are in ASCII-formatted text, the script encloses the configuration stored in the `$config` variable within the `configuration-text` tag element and stores the result in the value associated with the `queryargs` hash key `config-text`.

If the -t command-line option was not included when the **edit_configuration.pl** script was invoked, the file referenced by the $xmlfile variable contains Junos XML configuration tag elements. In this case, the script stores just the $config variable as the value associated with the queryargs hash key config.

```
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration text> . $config . </configuration-text>';
} else {
  $queryargs{'config'} = $config;
```

Editing the Configuration Data

The script invokes the edit_config method to load the configuration changes onto the device. It invokes the graceful_shutdown subroutine if the response from the NETCONF server has errors.

```


$res = $jnx->edit_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    # Get the error
    my $error = $jnx->get_first_error();
    get_error_info(%$error);
    # Disconnect
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
```

Committing the Configuration

If there are no errors up to this point, the script invokes the commit method to commit the configuration on the device and make it the active configuration.

```
# Commit the changes
print "Committing the <edit-config> changes ...\n";
$jnx->commit();
if ($jnx->has_error) {
    print "ERROR: Failed to commit the configuration.\n";
    graceful_shutdown($jnx, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Submit a Request to the NETCONF Server in Perl Client Applications | 454

Parse the NETCONF Server Response in Perl Client Applications

After establishing a connection to a NETCONF server, a NETCONF Perl client application can submit one or more requests by invoking Perl methods. The NETCONF server returns the appropriate information in an `<rpc-reply>` element. There are two ways of parsing the NETCONF server's response:

- By using functions of XML::LibXML::DOM
- By using functions of XML::LibXML::XPATHContext

For example, consider the following reply from a NETCONF server:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/20.1R1/junos" message-id='3'>
<chassis-inventory xmlns="http://xml.juniper.net/junos/20.1R1/junos-chassis">
<chassis style="inventory">
<name>Chassis</name>
<serial-number>G1234</serial-number>
<description>MX960</description>
...
</chassis>
</chassis-inventory>
</rpc-reply>
```

Suppose the user wants to parse the response and retrieve the value of the `<serial-number>` element.

The following code uses `XML::LibXMl::DOM` to retrieve the value. The example stores the response in a variable and calls methods of `DOM` to parse the response.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();

my $rpc = $jnx->get_dom();
my $serial = $rpc->getElementsByTagName("serial-number")->item(0)->getFirstChild->getData;
```

```
print ("\nserial number: $serial");
```

The following code uses `XML::LibXML::XPATHContext` to retrieve the value. The example stores the response in a variable and calls `XPathContext` methods to retrieve the value. The `local-name()` function returns the element name without the namespace. The XPATH expression appears on multiple lines for readability.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();

my $rpc= $jnx->get_dom();
my $xpc = XML::LibXML::XPathContext->new($rpc);
my $serial=$xpc->findvalue('
  /*[local-name()="rpc-reply"]
  /*[local-name()="chassis-inventory"]
  /*[local-name()="chassis"]
  /*[local-name()="serial-number"]');

print ("\nserial number: $serial");
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Submit a Request to the NETCONF Server in Perl Client Applications | 454

Close the Connection to the NETCONF Server in Perl Client Applications

In NETCONF Perl client applications, you can end the NETCONF session and close the connection to the device by invoking the `disconnect` method.

Several of the sample scripts included in the NETCONF Perl client distribution invoke the `disconnect` method in standalone statements. For example:

```
$jnx->disconnect();
```

The **edit_configuration.pl** sample script invokes the `graceful_shutdown` method, which takes the appropriate actions with regard to the configuration database and then invokes the `disconnect` method.

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

RELATED DOCUMENTATION

Write NETCONF Perl Client Applications | 444

Connect to the NETCONF Server in Perl Client Applications | 447

6

PART

YANG

CHAPTER 17

YANG Overview

IN THIS CHAPTER

Understanding YANG on Devices Running Junos OS

Yet Another Next Generation (YANG) is a standards-based, extensible data modeling language used to model configuration and operational state data, remote procedure calls (RPCs), and server event notifications of network devices. The NETMOD working group in the IETF originally designed YANG to model network management data and to provide a standard for the content layer of the Network Configuration Protocol (NETCONF) model. However, YANG is protocol-independent, and YANG data models can be used independent of the transport or RPC protocol and can be converted into any encoding format supported by the network configuration protocol.

Juniper Networks publishes YANG modules that define the configuration hierarchies, operational commands, operational state data, and YANG extensions for Junos devices. You can download the YANG modules from the Juniper Networks website or the Juniper Networks GitHub repository for YANG, or you can generate the modules on a Junos device.

YANG uses a C-like syntax, a hierarchical organization of data, and provides a set of built-in types as well as the capability to define derived types. YANG emphasizes readability, modularity, and flexibility, which is achieved through the use of modules (and submodules), reusable types, and node groups.

A YANG module defines a data model and determines the encoding for that data. A YANG module defines a data model through its data, and the hierarchical organization of and constraints on that data. A module can be a complete, standalone entity, or it can reference definitions in other modules and submodules as well as augment other data models with additional nodes.

A YANG module defines not only the syntax but also the semantics of the data. It explicitly defines relationships between and constraints on the data. You can create syntactically correct configuration data that meets constraint requirements and you can validate the data against the model before uploading it and committing it on a device.

YANG uses modules to define configuration and state data, notifications, and RPCs for network operations. This structure is similar to how the Structure of Management Information (SMI) utilizes MIBs to model data for SNMP operations. However, YANG has the benefit of being able to distinguish between operational and configuration data. YANG maintains compatibility with SNMP's SMIv2, and you can use `libsmi` to translate SMIv2 MIB modules into YANG modules and vice versa. Additionally, when you cannot use a YANG parser, you can translate YANG modules into YANG Independent Notation (YIN), which is an equivalent XML syntax that XML parsers and XSLT scripts can read.

You can use existing YANG-based tools or develop custom network management applications to utilize YANG modules for faster and more accurate network programmability. For example, a client application could leverage YANG modules to generate vendor-specific configuration data for different devices and validate that data before uploading it to the device. The application could also handle and troubleshoot unexpected RPC responses and errors.

For information about YANG, see RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, and related RFCs.

RELATED DOCUMENTATION

YANG Modules Overview | **479**

Use Juniper Networks YANG Modules | **519**

show system schema

Understanding Junos YANG Modules

IN THIS SECTION

Juniper Networks publishes the schema for Junos devices using YANG models for the configuration and operational state data, operational commands, and Junos extensions. The following sections discuss the native Junos YANG modules.

Junos YANG Modules Overview

Juniper Networks provides YANG modules that define the configuration hierarchies, operational commands and state data, as well as YANG extensions and types, for devices running Junos OS and devices running Junos OS Evolved. YANG modules are specific to a device family. Table 22 on page 472 outlines the identifiers for the different device families and indicates which platforms are in each family. Starting in Junos OS Evolved Release 23.4R2, all Junos OS Evolved platforms use the `junos` device family identifier.

Table 22: Junos Device Families

Device Family Identifier	Junos OS Platforms	Junos OS Evolved Platforms
junos	ACX Series EX Series (certain platforms) MX Series PTX Series	ACX Series PTX Series QFX Series (23.4R2 and later)
junos-es	SRX Series	–
junos-ex	EX Series (certain platforms)	–
junos-qfx	QFX Series	QFX Series (23.2 and earlier)

TIP: Different platforms within the same series might be categorized under different device families. You can verify the family for a specific device by executing the `show system information` operational mode command or the `<get-system-information/>` RPC on the device.

The value of the `Family` field in the command output or the `<os-name>` element in the RPC reply indicates the device family.

Table 23 on page 473 summarizes the YANG modules that are native to Junos devices. A Junos YANG module's name and filename include the device family, and when applicable, the area of the configuration or command hierarchy to which the schema in the module belongs. The module's filename also includes a revision date.

NOTE: Modules that do not require family-specific schemas and that are common to all platforms use the `junos` device family for the module's name, filename, and namespace.

Table 23: Juniper Networks Native YANG Modules

Junos YANG Module	Description	Module Name	Releases
Configuration modules	Define the schema for the Junos configuration hierarchy. The configuration YANG module comprises a root module (***family*-conf-root**) that is augmented by multiple smaller modules.	*`family`*`-conf-`*`hierarchy`*	17.4R1 and later
Operational command modules	Represent the operational command hierarchy and the collective group of modules that define the remote procedure calls (RPCs) for operational mode commands. There are separate modules for the different areas of the command hierarchy.	*`family`*`-rpc-`*`hierarchy`*	17.4R1 and later
`junos-state` state modules	Curated set of YANG modules for operational state data.	`junos-state-`*`area`*	22.2R1 and later
`genstate` state modules	Define YANG data models for operational state. The models expose a subset of show command data through the gNMI `subscribe` RPC. The `genstate` modules comprise a top-level root module augmented by modules for each operational state area.	`junos-genstate-`*`root-tag`*	24.2R1 and later (Junos OS Evolved) 25.4R1 and later (Junos OS)

Table 23: Juniper Networks Native YANG Modules *(Continued)*

Junos YANG Module	Description	Module Name	Releases
DDL extensions module	Contains Data Definition Language (DDL) statements for Junos devices. This module includes the `must` and `must-message` keywords, which identify configuration hierarchy constraints that use special keywords. The module also includes statements that are required in custom RPCs.	`junos-common-ddl-extensions`	17.4R1 and later
ODL extensions module	Contains Output Definition Language (ODL) statements that you can use to create and customize formatted ASCII output for RPCs executed on Junos devices.	`junos-common-odl-extensions`	17.4R1 and later
Metadata annotations extensions module	Defines metadata annotations for configuration operations. Annotations are defined in RFC 7952, *Defining and Using Metadata with YANG*.	`junos-configuration-metadata`	22.2R1 and later (Junos OS Evolved)
Types module	Contains definitions for YANG types.	`junos-common-types`	17.4R1 and later

To support YANG modules for different device families in different releases, the downloaded modules are organized by device family. Each module's name, filename, and namespace reflects the device family to which the schema in the module belongs.

For information about obtaining the modules, see "Download and Generate Junos YANG Modules " on page 474.

For information about the module namespaces, see "Understanding Junos YANG Module Namespaces and Prefixes" on page 476.

Download and Generate Junos YANG Modules

You can retrieve the Junos OS and Junos OS Evolved YANG modules by:

- Downloading the modules from the Juniper Networks website at https://www.juniper.net/support/downloads

- Downloading the modules from the Juniper/yang GitHub repository
- Generating the modules on a Juniper Networks device

NOTE: Starting in Junos OS Evolved Release 23.4R1, we publish the Junos OS Evolved native Yang modules on the Juniper Networks download site and on GitHub. In earlier releases, you must generate the modules on the device.

Junos YANG modules are specific to a device family. As a result, the download package and GitHub repository include a separate directory for each device family's modules. They also include a **common** directory for the modules that are common to all device families. Each family-specific directory uses its device family identifier as the directory name and contains the modules supported by the platforms in that family. The device family identifiers are defined in Table 22 on page 472.

The YANG modules generated on a local device, by default, contain family-specific schemas. Family-specific schemas are identical across all devices in the given device family. You can generate modules with device-specific schemas for the configuration and operational command modules. To emit device-specific schemas, configure the `device-specific` configuration statement at the `[edit system services netconf yang-modules]` hierarchy level.

Starting in Junos OS Evolved Release 23.4R1, we publish the Junos OS Evolved YANG modules on the Juniper Networks download site and on GitHub. In earlier releases, you must generate the modules on the device.

Additionally, starting in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1, we provide all YANG data models for a given OS and release in a single download package and GitHub repository folder. The package and repository folder include:

- Native configuration, state, and RPC data models
- OpenConfig configuration and state models supported by that OS
- IETF models supported by that OS

For more information about how to download or generate the Junos OS YANG modules, see "Use Juniper Networks YANG Modules" on page 519.

Understanding Junos YANG Module Namespaces and Prefixes

The Junos YANG modules use a namespace that includes the device family, the module type, and an identifier that is unique to each module. The identifier differentiates the namespace of the module from that of other modules. The namespace format is:

```
namespace "http://yang.juniper.net/device-family/type/identifier";
```

Where:

device-family Identifier for the device family to which the schema in the module belongs, for example, `junos`, `junos-es`, `junos-ex`, or `junos-qfx`. The different device families are outlined in Table 22 on page 472.

Modules with device-specific schemas and modules with family-specific schemas both use the same device family identifier in the namespace.

> **NOTE**: The common modules use the `junos` device family identifier in the namespace, but the modules are common to all device families.

identifier String that differentiates the namespace of the module from that of other modules.

Junos configuration and command modules include an identifier that indicates the area of the configuration or command hierarchy to which the schema in the module belongs. `Genstate` modules use an identifier that indicates the operational state area in the module. Common modules use the module name differentiator as an identifier, for example, `odl-extensions`.

type Type of the module. Possible values include:

- `conf`—Configuration YANG module that defines the schema for the indicated area of the configuration.
- `rpc`—Operational command YANG module that defines the RPCs for operational commands in the indicated area of the command hierarchy.
- `common`—Extension or type module that is common across all device families.
- `genstate`—YANG module that defines operational state data.

Table 24 on page 477 outlines each module's namespace URI and prefix (as defined by the module's `prefix` statement). The prefix for each configuration YANG module reflects the configuration statement hierarchy that is included in that module. Similarly, the prefix for each operational command module reflects the command hierarchy area of the RPCs included in that module. The Junos YANG extension and type modules use the `junos` device family identifier in the namespace, but the modules are common to all device families.

Table 24: Namespaces and Prefixes for Junos YANG Modules

YANG Module	Release	Namespace URI	Prefix
Configuration modules	17.4R1 and later	`http://yang.juniper.net/`*`device-family`*`/conf/`*`hierarchy`*	`jc` (root module) `jc-`*`hierarchy`*
Operational command modules	17.4R1 and later	`http://yang.juniper.net/`*`device-family`*`/rpc/`*`hierarchy`*	*hierarchy*
`junos-state` state modules	22.2R1 and later	`http://yang.juniper.net/junos/state/`*`state-area`*	`js-`*`area`*
`genstate` state models	24.2R1 and later	`http://yang.juniper.net/junos/genstate/`*`root-tag`*	`jgs` (root module)
DDL extensions module	17.4R1 and later	`http://yang.juniper.net/junos/common/ddl-extensions`	`junos`
ODL extensions module	17.4R1 and later	`http://yang.juniper.net/junos/common/odl-extensions`	`junos-odl`
Metadata annotations extensions module	22.2R1 and later	`http://yang.juniper.net/junos/jcmd`	`jcmd`
Types module	17.4R1 and later	`http://yang.juniper.net/junos/common/types`	`jt`

When you configure the `rfc-compliant` statement at the `[edit system services netconf]` hierarchy level and request configuration data in a NETCONF session, the server sets the default namespace for the `<configuration>` element to the same namespace as in the corresponding YANG model. For example:

```
<rpc>
  <get-config>
    <source>
```

```
        <running/>
      </source>
    </get-config>
  </rpc>

  <nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://
  xml.juniper.net/junos/25.2R1.8-EVO/junos">
  <nc:data>
  <configuration
      xmlns="http://yang.juniper.net/junos/conf/root"
      junos:commit-seconds="1763162210"
      junos:commit-localtime="2025-11-14 15:16:50 PST"
      junos:commit-user="admin">

  ...
  </configuration>
  </nc:data>
  </nc:rpc-reply>
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.4R2-EVO	Starting in Junos OS Evolved Release 23.4R2, native YANG modules for QFX Series devices use the `junos` device family identifier instead of `junos-qfx`.
23.4R1 and 23.4R1-EVO	Starting in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1, we provide all YANG data models for a given OS and release in a single download package and GitHub repository folder.
23.4R1-EVO	Starting in Junos OS Evolved Release 23.4R1, we publish the Junos OS Evolved native YANG modules on the Juniper Networks download site and on GitHub. In earlier releases, you must generate the modules on the device.
22.4R1 and 22.4R1-EVO	Starting in Junos OS Release 22.4R1 and Junos OS Evolved Release 22.4R1, YANG modules that define RPCs include the `junos:command` extension statement in schemas emitted with extensions.

RELATED DOCUMENTATION

Use Juniper Networks YANG Modules | **519**

Understanding the YANG Modules That Define the Junos OS Configuration | **480**

Understanding the YANG Modules for Junos Operational Commands | **484**

Understanding the Junos DDL Extensions YANG Module | **500**

show system schema

YANG Modules Overview

YANG data models comprise modules and submodules and can define configuration and state data, notifications, and RPCs for use by YANG-based clients. A YANG module defines a data model through its data, and the hierarchical organization of and constraints on that data. Each module is uniquely identified by a namespace URI.

A module defines a single data model. However, a module can reference definitions in other modules and submodules by using the `import` statement to import external modules or the `include` statement to include one or more submodules. Additionally, a module can augment another data model by using the `augment` statement to define the placement of the new nodes in the data model hierarchy and the `when` statement to define the conditions under which the new nodes are valid. A module uses the `feature` statement to specify parts of a module that are conditional and the `deviation` statement to specify where the device's implementation might deviate from the original definition.

When you import an external module, you define a prefix that is used when referencing definitions in the imported module. We recommend that you use the same prefix as that defined in the imported module to avoid conflicts.

YANG models data using a hierarchical, tree-based structure with nodes. YANG defines four nodes types. Each node has a name, and depending on the node type, the node might either define a value or contain a set of child nodes. The nodes types are:

- leaf node—Contains a single value of a specific type
- leaf-list node—Contains a sequence of leaf nodes
- container node—Contains a grouping of related nodes containing only child nodes, which can be any of the four node types
- list node—Contains a sequence of list entries, each of which is uniquely identified by one or more key leafs

In YANG, each leaf and leaf-list node includes the `type` statement to identify the data type for valid data for that node. YANG defines a set of built-in types and also provides the `typedef` statement for defining a derived type from a base type, which can be either a built-in type or another derived type.

By default, a node defines configuration data. A node defines state data if it is tagged as `config false`. Configuration data is returned using the NETCONF `<get-config>` operation, and state data is returned using the NETCONF `<get>` operation.

For detailed information about the syntax and semantics of the YANG language, see:

- RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*
- RFC 7950, *The YANG 1.1 Data Modeling Language*

RELATED DOCUMENTATION

Understanding YANG on Devices Running Junos OS | **470**

Use Juniper Networks YANG Modules | **519**

show system schema

Understanding the YANG Modules That Define the Junos OS Configuration

IN THIS SECTION

Overview of the Configuration YANG Modules

Juniper Networks publishes the Junos OS configuration schema using YANG models. The Junos OS configuration schema is published using a root configuration module that is augmented by multiple, smaller modules. This enables consumers of the schema to import only the modules required for their tasks.

The root configuration module comprises the top level configuration node and any nodes that are not emitted as separate modules. Separate, smaller modules augment the root configuration module for the different configuration statement hierarchies. These modules contain the schema for the configuration statement hierarchy level that is indicated by the module’s name, filename, and namespace.

The following example shows a portion of the YANG module that defines the schema for the `[edit interfaces]` statement hierarchy:

```
/*
 * Copyright (c) 2025 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-conf-interfaces {
   namespace "http://yang.juniper.net/junos/conf/interfaces";

   prefix jc-interfaces;

   import junos-common-types {
     prefix jt;
     revision-date 2024-01-01;
   }

   import junos-conf-root {
     prefix jc;
     revision-date 2024-01-01;
   }

   organization "Juniper Networks, Inc.";

   contact "yang-support@juniper.net";

   description "Junos interfaces configuration module";

   revision 2024-01-01 {
     description "Junos: 24.4R1.11";
   }

   augment /jc:configuration {
     uses interfaces-group;
   }

   augment /jc:configuration/jc:groups {
     uses interfaces-group;
   }
   ...
```

YANG utilities need to import only those modules required for the specific configuration task at hand. As a result, tools that consume the configuration modules require less time to compile, validate, or perform other functions on the modules than when importing a single, large module.

To determine the configuration YANG module corresponding to a specific area of the configuration, issue the `show | display detail` configuration mode command. In the following example, the schema for the `[edit protocols ospf]` hierarchy level is included in the **junos-conf-protocols@2024-01-01.yang** module.

```
user@host# show protocols ospf | display detail
##
## ospf: OSPF configuration
## YANG module: junos-conf-protocols@2024-01-01.yang
## package: junos-routing-ospf-advanced junos-routing-ospf-basic
##
##
## Area ID
## package: junos-routing-ospf-basic
##
area 0.0.0.0 {
...
```

How to Obtain the Configuration YANG Modules

You can download the Junos native YANG modules from the Juniper Networks download site or the Juniper/yang GitHub repository. You can also generate the modules on the local device.

To generate the configuration modules on the local device, issue the `show system schema format yang module` *`module`* command. Specify an individual module name to return a single configuration module, or specify `all-conf` to return all configuration modules.

```
user@host> show system schema format yang module all-conf output-directory /var/tmp/yang
```

If you specify `module all-conf`, the output files include both native Junos OS configuration modules as well as any standard or custom configuration modules that have been added to the device.

Starting in Junos OS Release 19.2R1, the `show system schema` command must include the `output-directory` command option and specify the directory in which to generate the files. In earlier releases, you can omit the `output-directory` option when requesting a single module to display the module in standard output.

NOTE: The native YANG modules generated on a local device contain family-specific schemas, which are identical across all devices in the given device family. To generate device-specific modules, configure the `device-specific` configuration statement at the `[edit system services netconf yang-modules]` hierarchy level.

NOTE: To generate YANG modules from a remote session, execute the `<get-yang-schema>` Junos OS RPC or the `<get-schema>` NETCONF operation with the appropriate options.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.4R1-EVO	Starting in Junos OS Evolved Release 23.4R1, native YANG modules for QFX Series devices use the `junos` device family identifier instead of `junos-qfx`.
22.4R1 and 22.4R1-EVO	Starting in Junos OS Release 22.4R1 and Junos OS Evolved Release 22.4R1, if a YANG leaf node is type identityref, Junos devices emit the namespace-qualified form of the identity in the JSON encoding of that node. Additionally, Junos devices accept the simple (no namespace) and the namespace-qualified form of an identity in JSON configuration data. In earlier releases, Junos devices only emit and accept the simple form of an identity.
19.2R1	Starting in Junos OS Release 19.1R2 and 19.2R1, the `show system schema` command must include the `output-directory` command option and specify the directory in which to generate the file or files.

RELATED DOCUMENTATION

Use Juniper Networks YANG Modules | 519

Understanding Junos YANG Modules | 471

show system schema

Understanding the YANG Modules for Junos Operational Commands

IN THIS SECTION

Overview of the Operational Command YANG Modules

Juniper Networks publishes YANG modules that define the remote procedure calls (RPCs) for Junos operational mode commands. Due to the large number of operational commands, each device family has multiple YANG modules. Each top-level operational command group (`clear`, `file`, `monitor`, and so on) has a module when that hierarchy has at least one command with an RPC equivalent. Additionally, the `show` command hierarchy has a separate module for each area within that hierarchy.

The operational command modules define the RPCs corresponding to the operational commands in the area indicated by the module's filename. The following example shows a portion of the **junos-rpc-clear@2024-01-01.yang** module, which contains the RPCs for commands in the `clear` command hierarchy:

```
/*
 * Copyright (c) 2025 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-rpc-clear {
   namespace "http://yang.juniper.net/junos/rpc/clear";

   prefix clear;

   import junos-common-types {
     prefix jt;
     revision-date 2024-01-01;
   }

   organization "Juniper Networks, Inc.";

   contact "yang-support@juniper.net";
```

```
description "Junos RPC YANG module for clear command(s)";

revision 2024-01-01 {
  description "Junos: 24.4R1.11";
}

rpc clear-cli-logical-system {
  description "Clear logical system association";
  output {
    choice output_c {
      case output-tag {
        leaf output {
          type string;
        }
      }
      case multichassis-tag {
        anyxml multi-routing-engine-results;
      }
    }
  }
}
rpc clear-cli-tenant {
  description "Clear teannt association";
  output {
    choice output_c {
      case output-tag {
        leaf output {
          type string;
        }
      }
      case multichassis-tag {
        anyxml multi-routing-engine-results;
      }
    }
  }
}
...
```

How to Obtain the Operational Command YANG Modules

You can download the Junos native YANG modules from the Juniper Networks download site or the Juniper/yang GitHub repository. You can also generate the modules on the local device.

To generate the operational command YANG modules on the local device issue the `show system schema format yang module` *`module`* command. Specify an individual module name to return a single operational command module, or specify `all-rpc` to return all operational command modules.

```
user@host> show system schema format yang module all-rpc output-directory /var/tmp/yang
```

If you specify `module all-rpc`, the output files include both native Junos operational command modules as well as any standard or custom operational command modules that have been added to the device. To use an RPC in your custom YANG module, you must import the module that contains the RPC into your custom module.

Starting in Junos OS Release 19.2R1, the `show system schema` command must include the `output-directory` command option and specify the directory in which to generate the files. In earlier releases, you can omit the `output-directory` option when requesting a single module to display the module in standard output.

NOTE: The native YANG modules generated on a local device contain family-specific schemas, which are identical across all devices in the given device family. To generate device-specific modules, configure the `device-specific` configuration statement at the `[edit system services netconf yang-modules]` hierarchy level.

NOTE: To generate YANG modules from a remote session, execute the `<get-yang-schema>` Junos OS RPC or the `<get-schema>` NETCONF operation with the appropriate options.

You can emit the YANG schemas with additional Junos extension statements. The Junos extensions are defined in "Understanding the Junos DDL Extensions YANG Module" on page 500. To include extensions, configure the `emit-extensions` statement at the `[edit system services netconf yang-modules]` hierarchy level. The device emits the `junos:command` extension statement starting in Junos OS Release 22.4R1 and Junos OS Evolved Release 22.4R1.

Understanding the RPC Output Schema

YANG defines operations using the `rpc` statement. The RPC definition can include `input` and `output` substatements that describe the operation's input and output parameters. Starting in Junos OS Release

23.1R1 and Junos OS Evolved Release 23.2R1, the RPC's `output` statement includes an accurate output schema, and Junos devices emit the new schemas by default.

In earlier releases, the RPC's output schema includes the `anyxml` statement to represent an unknown chunk of XML in the RPC reply. To generate the earlier RPC schemas containing the `anyxml` statement on a Junos device, configure the `device-specific` and `emit-anyxml-in-rpc-output` statements at the `[edit system services netconf yang-modules]` hierarchy level.

```
[edit system services netconf yang-modules]
user@host# set device-specific
user@host# set emit-anyxml-in-rpc-output
user@host# commit
```

After you configure the statements, the `show system schema` command generates the schemas that use `anyxml`.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.4R1-EVO	Starting in Junos OS Evolved Release 23.4R1, native YANG modules for QFX Series devices use the `junos` device family identifier instead of `junos-qfx`.
23.2R1-EVO	Starting in Junos OS Evolved Release 23.2R1, the YANG modules that define Junos RPCs include accurate output schemas.
23.1R1	Starting in Junos OS Release 23.1R1, the YANG modules that define Junos RPCs include accurate output schemas.
22.4R1 and 22.4R1-EVO	Starting in Junos OS Release 22.4R1 and Junos OS Evolved Release 22.4R1, YANG modules that define RPCs include the `junos:command` extension statement in schemas emitted with extensions.
19.2R1	Starting in Junos OS Release 19.1R2 and 19.2R1, the `show system schema` command must include the `output-directory` command option and specify the directory in which to generate the file or files.

RELATED DOCUMENTATION

Use Juniper Networks YANG Modules | 519

Understanding Junos YANG Modules | 471

show system schema

Junos Genstate YANG Data Models

SUMMARY

The Juniper Networks `genstate` YANG schema defines YANG data models for operational state on Junos devices. gRPC Network Management Interface (gNMI) clients can subscribe to the resource paths defined in the models to request state data.

IN THIS SECTION

Genstate YANG Data Models Overview

IN THIS SECTION

Starting in Junos OS Evolved Release 24.2R1, Juniper Networks publishes the *genstate* YANG data models. The `genstate` models are subscribable YANG models for operational state data on Junos devices. They are a YANG representation of the operational command output. The `genstate` models comprise a top-level module augmented by modules for each of the different operational state areas as they are published and made available.

Junos devices have a rich set of native state data. You can retrieve operational state information from Junos devices by executing operational commands or Junos XML RPCs on the device. However, you must still parse the command output to extract specific data. Juniper also provides the curated `junos-`

state YANG data models that telemetry collectors can consume, but the models include only a subset of operational areas and states.

The genstate YANG models expose state data available in operational show commands through the gNMI subscribe RPC. The modules describe the resource paths that correspond to specific state data on the target network device. gNMI telemetry collectors can subscribe to a resource path in the published YANG models to query for the state data for that instance.

gNMI clients in gRPC dial-in environments can subscribe to the genstate published paths. A client can use STREAM subscriptions in SAMPLE mode to request the data. ON_CHANGE mode is not supported. For information about using gNMI to subscribe to telemetry data, see the Junos Telemetry Interface User Guide.

The genstate YANG data models are published and updated in different releases. You can check the available models and supported paths for a given release in the following ways:

- View or download the modules from the Juniper/yang GitHub repository.
- Download the modules from the Juniper Networks website at https://www.juniper.net/support/downloads.
- Emit the modules on devices running Junos OS or devices running Junos OS Evolved.

Benefits of the Genstate YANG State Models

- Increase the surface area of operational state available through gNMI and thus enable you to make more informed usage decisions about the device and network when you use gNMI to monitor state.
- Simplify how you monitor device state by enabling you to move toward a single northbound interface.

Genstate Modules Overview

The genstate YANG data models comprise a top-level root module augmented by modules for each of the available operational state areas. The genstate schema uses origin 'genstate'.

The top-level module is as follows:

```
module junos-genstate {
  namespace "http://yang.juniper.net/junos/genstate/";
  prefix jgs;

  organization
    "Juniper Networks, Inc.";
  contact
    "Juniper Networks, Inc.

     1133 Innovation Way
     Sunnyvale, CA

     +1 888 314-5822

     E-mail: yang-support@juniper.net";

  description
    "This module contains a collection of top level nodes for JUNOS genstate data.

     Copyright (c) 2023 Juniper Networks, Inc.
     All rights reserved.";

  revision 2023-01-01 {
    description "Junos: ";
  }

  grouping genstate-top {
    description "Top-level grouping for JUNOS genstate data";
    container genstate {
      description "Encapsulating top-level state container for all JUNOS genstate data";
      config false;
    }
  }
  uses genstate-top;
}
```

The top-level `genstate` module is augmented by the modules published for each operational state area. When you issue operational commands or RPCs on a Junos device, the device returns the XML output enclosed in a top-level element. The root-level tag name describes the enclosed data. For example, `show interfaces` commands return XML output that is enclosed in an `<interface-information>` tag. The `genstate` modules include this root tag name in the module name and filename to easily identify the data described in the module. The module prefix is based on the root tag name and so varies for each module.

For example, the `junos-genstate-interface-information` module describes the data that would normally be included in the `<interface-information>` element in Junos command and RPC output. Thus, the model defines the resource paths that are available for subscription by telemetry collectors for interface state data.

```
module junos-genstate-interface-information {
  namespace "http://yang.juniper.net/junos/genstate/interface-information";
  prefix jgii;

  import junos-genstate {
    prefix jgs;
    revision-date 2023-01-01;
  }

  import junos-common-types {
    prefix jt;
    revision-date 2023-01-01;
  }

  organization
    "Juniper Networks, Inc.";

  contact
    "Juniper Networks, Inc.

     1133 Innovation Way
     Sunnyvale, CA

     +1 888 314-5822

     E-mail: yang-support@juniper.net";

  description "Junos genstate data model for interface-information";

  revision 2023-01-01 {
    description "Junos: ";
  }

  grouping interface-information-top {
    description "Top-level grouping";
    container interface-information {
      config false;
      description "Top-level container";
      list physical-interface {
        leaf name {
          type string;
          description "Name of this item";
        }
        leaf oper-status {
          type string;
          description "Current operational state of the interface";
        }
        leaf local-index {
          type int32;
          description "Local kernel index for this interface";
        }
        leaf snmp-index {
          type int32;
          description "SNMP ifIndex for this interface";
        }
        container if-config-flags {
          leaf iff-none {
            type empty;
          }
          leaf iff-hardware-down {
            type empty;
          }
          leaf iff-down {
            type empty;
          }
          leaf iff-up {
            type empty;
          }
          leaf iff-admin-down {
            type empty;
          }
          leaf iff-admin-up {
            type empty;
          }
          leaf iff-link-down {
            type empty;
          }
          leaf iff-device-down {
            type empty;
          }
          leaf iff-point-to-point {
            type empty;
            }
            leaf iff-point-to-multipoint {
              type empty;
            }
            leaf plp-to-clp {
              type empty;
            }
            leaf iff-multiaccess {
              type empty;
            }
            leaf iff-snmp-traps {
              type empty;
            }
            leaf iff-looped {
              type empty;
            }
            leaf iff-framing-conflict {
              type empty;
            }
            leaf internal-flags {
              type string;
              description "Hexadecimal value of internal flag bits";
            }
          }
          list logical-interface {
            leaf name {
              type string;
              description "Name of this item";
            }
            leaf local-index {
              type int32;
              description "Local kernel index for this interface";
            }
            leaf snmp-index {
              type int32;
              description "SNMP ifIndex for this interface";
            }
            leaf generation {
              type string;
              description "Generation number used to distinguish between successive instances of
this interface";
            }
            leaf description {
```

```
          type string;
          description "Description of this interface";
        }
        leaf link-address {
          type string;
          description "Link address on this logical interface";
        }
        leaf encapsulation {
          type string;
          description "Encapsulation on the logical interface";
        }
        leaf subunit {
          type int32;
          description "Subunit for this interface";
        }
        container if-config-flags {
          leaf iff-none {
            type empty;
          }
          leaf iff-hardware-down {
            type empty;
          }
          leaf iff-down {
            type empty;
          }
          leaf iff-up {
            type empty;
          }
          leaf iff-admin-down {
            type empty;
          }
          leaf iff-admin-up {
            type empty;
          }
          leaf iff-link-down {
            type empty;
          }
          leaf iff-device-down {
            type empty;
          }
          leaf iff-point-to-point {
            type empty;
          }
```

```
            leaf iff-point-to-multipoint {
              type empty;
            }
            leaf plp-to-clp {
              type empty;
            }
            leaf iff-multiaccess {
              type empty;
            }
            leaf iff-snmp-traps {
              type empty;
            }
            leaf iff-looped {
              type empty;
            }
            leaf iff-framing-conflict {
              type empty;
            }
            leaf internal-flags {
              type string;
              description "Hexadecimal value of internal flag bits";
            }
          }
          leaf admin-status {
            type string;
            description "Desired state of the interface";
          }
          leaf oper-status {
            type string;
            description "Current operational state of the interface";
          }
        }
      }
    }
  }
  augment "/jgs:genstate" {
    description "Adds interface-information to top-level genstate";
    uses interface-information-top;
  }
}
```

How to Construct genstate Resource Paths

A telemetry collector can subscribe to the different resource paths as defined in the `genstate` models to query for state data on devices that support this feature. The `genstate` model uses the following syntax:

```
origin:/root/operational-response-tag/optional-child-tags-to-resource
```

The path uses the origin `genstate`, a root tag named `genstate`, and a top-level tag name for the operational state area. For example, to subscribe to `genstate` resource paths for all `interface-information` state data, you would use the following path:

```
genstate:/genstate/interface-information
```

You can include supported child tags to retrieve state data for a specific resource. For example, a gNMI client can use the following path to retrieve the operational state for all interfaces:

```
genstate:/genstate/interface-information/physical-interface/oper-status
```

The `genstate` resource paths can use simple path expressions to query data. You can use path-based and key-based filters. Table 25 on page 496 outlines the supported expressions and provides some sample resource paths. Other query patterns are not supported. The path expressions support container, leaf, and key-based filtering and align with gNMI conventions, as outlined in Schema path encoding conventions for gNMI.

Table 25: Path Expressions Supported in genstate Resource Paths

Filter Type	Example Paths
Container	• `genstate:/genstate/interface-information` • `genstate:/genstate/snmp-statistics/snmp-input-statistics`
Leaf	• `genstate:/genstate/commit-revision-information/revision` • `genstate:/genstate/lldp/lldp-global-status`

Table 25: Path Expressions Supported in genstate Resource Paths *(Continued)*

Filter Type	Example Paths
List key	• `genstate:/genstate/interface-information/physical-interface[name=re0:mgmt-0]` • `genstate:/genstate/interface-information/physical-interface[name=re0:mgmt-0]/oper-status` • `genstate:/genstate/interface-information/physical-interface[name=et-1/0/1]/speed`
Multiple list keys	• `genstate:/genstate/interface-information/physical-interface[name=et-1/0/1]/logical-interface[name=et-1/0/1.16386]/local-index` • `genstate:/genstate/interface-information/physical-interface[name=lo0]/logical-interface[name=lo0.0]/address-family[address-family-name=inet]/address-family-name`
Wildcard list key	• `genstate:/genstate/interface-information/physical-interface[name=*]/oper-status` • `genstate:/genstate/interface-information/physical-interface/oper-status`

Path-based filters can select containers, lists, or leaf nodes. Key-based filters select a subset of list elements and support using precise values, specifying wildcards, or omitting the value. Specifying a wildcard or omitting a list key value matches all entries within a particular list.

We recommend that you always enclose the path in quotation marks. This approach prevents parsing errors in most environments and handles paths with spaces or other special characters.

For example, the following command uses quotation marks around the entire path:

```
$ gnmic sub --mode stream --stream-mode sample --sample-interval 30s -a 198.51.100.1:32767 -u
grpc-user -p secret --tls-ca certs/serverRootCA.crt --path "genstate:/genstate/license-summary-
information/license-detail-usage-summary/feature-summary[name=FIB Scale]"
```

Map Genstate Model Resource Paths to CLI Commands

You can verify the CLI command that generates the output corresponding to a specific `genstate` resource path. To retrieve the command, use the `show system data-models genstate` operational command. Include the `xpath-cli-command-mapping` option and provide the path to the desired resource.

```
user@host> show system data-models genstate xpath-cli-command-mapping resource-path
```

The following example retrieves the command that generates the state data corresponding to the `genstate:/genstate/system-information/os-version` resource path.

```
user@host> show system data-models genstate xpath-cli-command-mapping genstate:/genstate/system-
information/os-version

Genstate xpath to CLI command mapping information (sometimes multiple commands might be mapped):

> show system information
```

Similarly, the following example maps the given resource path to the CLI command that includes that information in the output.

```
user@host> show system data-models genstate xpath-cli-command-mapping genstate:/genstate/
interface-information/physical-interface/oper-status

Genstate xpath to CLI command mapping information (sometimes multiple commands might be mapped):

> show interfaces detail
```

You can include list key values in the genstate path. When you issue the command or corresponding RPC and specify a list key in the path, enclose the list key value in single quotation marks and enclose the path in double quotation marks. The following examples request the command for genstate paths that specify a key.

```
user@host> show system data-models genstate xpath-cli-command-mapping "genstate:/genstate/
interface-information/physical-interface[name='et-1/0/1']/speed"

Genstate xpath to CLI command mapping information (sometimes multiple commands might be mapped):

> show interfaces et-1/0/1 brief
```

```
user@host> show system data-models genstate xpath-cli-command-mapping "genstate:/genstate/
license-summary-information/license-detail-usage-summary/feature-summary[name='FIB Scale']"

Genstate xpath to CLI command mapping information (sometimes multiple commands might be mapped):

> show system license detail
```

Similarly, you can use the `<get-genstate-xpath-cli-command-mapping>` RPC to retrieve the same information.

```
<rpc>
  <get-genstate-xpath-cli-command-mapping>
     <xpath>genstate:/genstate/interface-information/physical-interface[name='et-1/0/1']/speed</
xpath>
  </get-genstate-xpath-cli-command-mapping>
</rpc>
```

If a device does not support a particular genstate path, the command returns the following message:

```
error: No command mapping information found for xpath xpath.
The xpath does not seem to be exposed in genstate;
```

NOTE: If you do not have permission to execute a CLI command, you cannot access the corresponding *genstate* subscription path. You can only subscribe to *genstate* paths of commands for which you have valid permissions.

How to Obtain the genstate YANG Modules

You can download Junos native YANG modules from the Juniper Networks download site or the Juniper/yang GitHub repository. You can also generate the modules on the local device. For instructions on downloading the modules, see "Use Juniper Networks YANG Modules" on page 519.

To emit the genstate YANG module files on the local device issue the `show system schema format yang module` *`module`* command. Specify an individual module name to return a single genstate module, or specify `all-genstate` to return all genstate modules. You must also specify an output directory for the files.

```
user@host> show system schema format yang module module-name output-directory output-directory
```

For example, the following commands create a directory, write all the genstate modules to that directory, and view the directory listing.

```
user@host> file make-directory /var/tmp/yang
user@host> show system schema format yang module all-genstate output-directory /var/tmp/yang
user@host> file list /var/tmp/yang
/var/tmp/yang:
junos-genstate-arp-table-information@2025-01-01.yang
junos-genstate-bgp-diagnostics-overview@2025-01-01.yang
```

```
junos-genstate-bgp-diagnostics-warnings@2025-01-01.yang
junos-genstate-bgp-fabric-advertise@2025-01-01.yang
...
```

NOTE: To generate the YANG modules from a remote session, execute the `<get-yang-schema>` Junos OS RPC or the `<get-schema>` NETCONF operation with the appropriate options.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
25.4R1 & 25.4R1-EVO	Starting in Junos OS Release 25.4R1 and Junos OS Evolved Release 25.4R1, you can use the `show system schema` operational command or equivalent RPC to view the genstate YANG data models on a device.

Understanding the Junos DDL Extensions YANG Module

IN THIS SECTION

The Junos Data Definition Language (DDL) extensions YANG module contains YANG extensions for Junos devices. These extensions include statements that can define constraints on configuration data and the valid values for strings. There are also statements that you include in custom RPCs to define a CLI command for the RPC and to specify details about the action script to invoke when the RPC is executed. In addition, there are statements that you can use to define helper action scripts for individual command options and configuration statements, for example, to display a list of acceptable values for options or statements.

Table 26 on page 501 outlines the statements in the DDL extensions module and provides a brief description of each statement.

Table 26: Statements in the junos-extension Module

Statement Keyword	Argument Description
`action-execute`	Define the actions taken when you execute a custom RPC. Use the `script` substatement to define the RPC's action script, which is invoked when you execute the RPC.
`action-expand`	Define the script that calculates and displays the possible values for a given command option or configuration statement in a custom YANG data model when a user requests context-sensitive help in the CLI. Use the `script` substatement to define the Python script that implements the logic.
`cli-feature`	Identify certain CLI properties associated with some command options and configuration statements. See "cli-feature Supported Properties" on page 502.
`command`	String defining the operational command that is used to execute the corresponding RPC in the Junos OS CLI.
`must`	String that identifies a constraint on the configuration data. Whereas the argument for the YANG `must` statement is a string containing an XPath expression, the argument for the `junos:must` extension statement is a string containing special Junos OS syntax required for the expression of the configuration statement path. This might include special keywords such as `any`, `all`, and `unique`.
`must-message`	String that defines the warning message that is emitted when the constraint defined by the corresponding `junos:must` statement evaluates to false.
`pattern-message`	String that defines the error message emitted when the constraint defined by the corresponding `posix-pattern` statement evaluates to false.
`posix-pattern`	Restrict the values accepted for nodes of type string to those that match the POSIX regular expression defined in this string.
`script`	String specifying the name of an action script. This is a substatement of the `action-execute` or `action-expand` statement.

cli-feature Supported Properties

The `cli-feature` YANG extension identifies certain CLI properties associated with some command options and configuration statements. This extension is beneficial when a client consumes YANG data models, but for certain workflows, the client needs to generate CLI-based tools. Supported properties include:

- `homogenous`—Text-formatted configuration data groups list objects into a single set of braces under the object keyword.
- `nokeyword`—The CLI does not require explicitly specifying the leaf name in the input syntax.
- `oneliner/oneliner-plus`—Text-formatted configuration data places an object's attributes on a single line. The `oneliner` flag does so without exception; the `oneliner-plus` flag does so only when zero or one value occurs for an attribute.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
22.3R1 and 22.3R1-EVO	Starting in Junos OS Release 23.1R1 and Junos OS Evolved Release 23.1R1, YANG modules that define the configuration or RPCs include the `cli-feature` extension statements, where applicable, in schemas emitted with extensions.

RELATED DOCUMENTATION

Understanding Junos YANG Modules | **471**

Use Juniper Networks YANG Modules | **519**

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

YANG Metadata Annotations for Junos Devices

SUMMARY

Junos devices support YANG extensions that define metadata annotations, which you can use to perform specific operations on the Junos configuration.

IN THIS SECTION

Junos devices support YANG extensions to annotate instances of YANG data nodes with metadata. You can use the following extensions on supported devices:

- `junos-configuration-metadata`—Juniper annotations that you can use to perform specific configuration operations.
- `openconfig-metadata`—Annotations defined by the OpenConfig working group.

YANG metadata annotations and their corresponding JSON and XML encoding are defined in RFC 7952, *Defining and Using Metadata with YANG*. The `ietf-yang-metadata` module defines the YANG extension `annotation`.

NOTE: YANG metadata annotations should not be confused with Junos configuration annotations, which are comments that are included in the configuration, for example, by using the `annotate` configuration mode command.

junos-configuration-metadata Module Overview

The Juniper Networks `junos-configuration-metadata` module defines metadata annotations that enable you to perform specific operations on the Junos configuration.

```
user@host> show system schema module junos-configuration-metadata output-directory /var/tmp
user@host> file show /var/tmp/junos-configuration-metadata.yang
/*
 * junos-configuration-metadata.yang -- Defines annotations (RFC 7952) for
 * Junos configuration metadata operations.
 *
 * Copyright (c) 2021, Juniper Networks, Inc.
 * All rights reserved.
 */
module junos-configuration-metadata {
    namespace "http://yang.juniper.net/junos/jcmd";
    prefix "jcmd";

    import ietf-yang-metadata {
        prefix "md";
    }

    organization
        "Juniper Networks, Inc.";

    contact
        "yang-support@juniper.net";

    description
        "This Yang module defines annotations (RFC 7951) for Junos configuration
        metadata operations.";

    revision 2021-09-01 {
        description
            "Initial version.";
    }

    md:annotation active {
        type boolean;
        description
            "This annotation can be used in configuration XML/JSON to
            deactivate/activate a configuration element. Specifying the value
            'false' deactivates the configuration element. Specifying the
            value 'true' activates the configuration element. When the
            configuration element is deactivated and committed, the element
            remains in the configuration, but the element does not affect the
            functioning of the device.";
    }

    md:annotation protect {
        type boolean;
        description
            "This annotation can be used in configuration XML/JSON to
            protect/unprotect the configuration hierarchies and statements.
            Specifying the value 'true' protects the configuration
            hierarchy/statement. Specifying the value 'false' unprotects the
            configuration hiearchy/statement. The protect operation
            prevents changes to selected (protected) configuration hierarchies
            and statements.";
    }

    md:annotation comment {
        type string;
        description
            "This annotation must be used in configuration XML/JSON to
            add comments to a configuration element. To remove the existing
            comment, empty string has to be supplied as a value for this
            annotation.";
    }
}
```

Devices that support the `junos-configuration-metadata` annotations advertise the following capabilities in the NETCONF capabilities exchange:

```
<capability>http://yang.juniper.net/junos/jcmd?module=junos-configuration-
metadata&amp;revision=2021-09-01</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-yang-metadata?module=ietf-yang-
metadata&amp;revision=2016-08-05</capability>
```

Table 27 on page 506 outlines the `junos-configuration-metadata` annotations. The annotations use the `http://yang.juniper.net/junos/jcmd` namespace URI and the `jcmd` namespace prefix.

Table 27: junos-configuration-metdata Annotations

Annotation	Value	Description
`active`	`false`	Deactivate the specified configuration statement. The statement remains in the configuration but does not affect the device's operation.
	`true`	Activate the specified configuration statement. Use this annotation to activate a statement that was previously deactivated.
`comment`	*`string`*	Add a comment with additional information about the specified configuration statement, or remove an existing comment by setting the value to an empty string ("").
`protect`	`false`	Remove any previously applied `protect` state from the specified configuration statement and allow changes to that statement.
	`true`	Prevent future modifications to the specified statement, until such time that the `protect` state is removed.

Using `junos-configuration-metadata` Annotations in Configuration Data

You can use the `junos-configuration-metadata` annotations in a YANG-compliant NETCONF session to perform specific metadata operations on the configuration. Supported operations include adding comments to the configuration, deactivating or activating configuration hierarchies and statements, and protecting configuration hierarchies and statements, as described in the following sections:

- "Add Comments in the Configuration" on page 507
- "Activate or Deactivate Configuration Statements" on page 509
- "Protect or Unprotect Configuration Statements" on page 513

You can apply `junos-configuration-metadata` annotations on a container (statement hierarchy), leaf-list, leaf statement, or a list item (statement with an identifier). When you apply the annotations on leaf-list statements, you can only apply them at the leaf-list level, not on individual leaf-list entries.

You can use the YANG annotations in JSON or XML configuration data, as outlined in Table 28 on page 507. You can use the NETCONF `<edit-config>` operation to load XML configuration data, and you can use the Junos XML protocol `<load-configuration>` operation to load JSON or XML configuration data on a device.

Table 28: Using Configuration Metadata Annotations

Encoding	Syntax	Example
JSON (metadata object)	"*module-name*:*annotation*" : "*value*"	`"junos-configuration-metadata:comment" : "`*comment string*`"`
XML (XML attributes)	`xmlns:`*prefix*=*namespace-uri* *prefix*:*annotation*="*value*"	`<`*element-name* `xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:comment="`*comment string*`">`

Add Comments in the Configuration

IN THIS SECTION

You can use the `comment` annotation to add comments to a configuration statement. The following sections outline how to add a comment when loading JSON or XML configuration data.

JSON

To add a comment when loading JSON configuration data, include the `junos-configuration-metadata:comment` annotation in the metadata object for that statement and specify the comment as a string. To remove a comment, include an empty string ("").

The following example associates one comment with a hierarchy, another comment with a list entry that requires an identifier, and a third comment with an existing leaf statement.

```
<rpc>
<load-configuration format="json">
<configuration-json>
{
   "configuration" : {
      "protocols" : {
         "ospf" : {
            "@" : {
               "junos-configuration-metadata:comment" : "/* OSPF comment */"
            },
            "area" : [
            {
               "name" : "0.0.0.0",
               "interface" : [
               {
                  "@" : {
                     "junos-configuration-metadata:comment" : "/* From jnpr1 \n to jnpr2 */"
                  },
                  "name" : "et-0/0/1.0",
                  "@hello-interval" : {
                     "junos-configuration-metadata:comment" : "# set by admin"
                  }
               }
               ]
            }
            ]
         }
      }
   }
}
</configuration-json>
</load-configuration>
</rpc>
]]>]]>
```

XML

To add a comment when loading XML configuration data, include the `jcmd:comment` annotation as an XML attribute in the opening tag of that configuration element and specify the comment as a string. To remove a comment, include an empty string ("").

The following example associates one comment with a hierarchy, another comment with a list entry that requires an identifier, and a third comment with a leaf statement.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <protocols>
          <ospf xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:comment="/* OSPF comment
*/">
            <area>
              <name>0.0.0.0</name>
              <interface xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:comment="/* From
jnpr1 \n to jnpr2 */">
                <name>et-0/0/1.0</name>
                <hello-interval xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:comment="#
set by admin">5</hello-interval>
              </interface>
            </area>
          </ospf>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

Activate or Deactivate Configuration Statements

IN THIS SECTION

You can use the `active` annotation to deactivate a configuration statement or to activate a configuration statement that was previously deactivated. To deactivate a statement, set `active` to `false`. To activate a statement, set `active` to `true`.

The following sections outline how to deactivate and activate configuration statements in JSON and XML configuration data.

JSON

To deactivate or reactivate a configuration object in JSON, include the `"junos-configuration-metadata:active" : (false | true)` annotation in the metadata object for that statement.

```
<configuration-json>
{
    "configuration" : {
        /* JSON objects for parent levels */
            "@leaf-list-statement-name" : {
                "junos-configuration-metadata:comment" : "/* activate or deactivate a leaf-list
*/",
                "junos-configuration-metadata:active" : (false | true)
            },
            "level-or-container" : {
                "@" : {
                    "junos-configuration-metadata:comment" : "/* activate or deactivate a
hierarchy */",
                    "junos-configuration-metadata:active" : (false | true)
                },
                "object" : [
                {
                    "@" : {
                        "junos-configuration-metadata:comment" : "/* activate or deactivate an
object with an identifier */",
                        "junos-configuration-metadata:active" : (false | true)
                    },
                    "name" : "identifier",
                    "@statement-name" : {
                        "junos-configuration-metadata:comment" : "/* activate or deactivate a
statement */",
                        "junos-configuration-metadata:active" : (false | true)
                    }
                }
                ]
            }
        /* closing braces for parent levels */
    }
}
</configuration-json>
```

For example, the following RPC deactivates the `[edit protocols isis]` hierarchy, activates the `apply-groups` leaf-list statement, and modifies the specified event policy to deactivate the event-script action and reactivate the raise-trap action.

```
<rpc>
<load-configuration format="json">
<configuration-json>
{
    "configuration" : {
        "@apply-groups" : {
            "junos-configuration-metadata:active" : true
        },
        "protocols" : {
            "isis" : {
                "@" : {
                    "junos-configuration-metadata:active" : false
                }
            }
        },
        "event-options" : {
            "policy" : [
            {
                "name" : "raise-trap-on-ospf-nbrdown",
                "then" : {
                    "event-script" : [
                    {
                        "@" : {
                            "junos-configuration-metadata:active" : false
                        },
                        "name" : "ospf.xsl"
                    }
                    ],
                    "@raise-trap" : {
                        "junos-configuration-metadata:active" : true
                    }
                }
            }
            ]
        }
    }
}
</configuration-json>
```

```
</load-configuration>
</rpc>
```

XML

To deactivate or reactivate a configuration object, include the `jcmd:active="false"` or `jcmd:active="true"` annotation, respectively, as an XML attribute in the opening tag of that configuration element.

The following RPC deactivates the `[edit protocols isis]` hierarchy, activates the `apply-groups` leaf-list statement, and modifies the specified event policy to deactivate the event-script action and reactivate the raise-trap action.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <apply-groups xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:active="true"/>
        <protocols>
           <isis xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:active="false"/>
        </protocols>
        <event-options>
          <policy>
            <name>raise-trap-on-ospf-nbrdown</name>
            <then>
              <event-script xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:active="false">
                <name>ospf.xsl</name>
              </event-script>
              <raise-trap xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:active="true"/>
            </then>
          </policy>
        </event-options>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

Protect or Unprotect Configuration Statements

IN THIS SECTION

You can protect selected Junos configuration hierarchies and statements to prevent changes to those statements until such time that the protect attribute is removed.

The following sections outline how to protect or unprotect configuration statements in JSON and XML configuration data.

JSON

To protect or unprotect a configuration object in JSON, include the `"junos-configuration-metadata:protect" : (true | false)` annotation in the metadata object for that statement.

```
<configuration-json>
{
    "configuration" : {
        /* JSON objects for parent hierarchies */
            "@leaf-list-statement-name" : {
                "junos-configuration-metadata:comment" : "/* protect a leaf-list */",
                "junos-configuration-metadata:protect" : (false | true)
            },
            "hierarchy" : {
                "@" : {
                    "junos-configuration-metadata:comment" : "/* protect a hierarchy */",
                    "junos-configuration-metadata:protect" : (false | true)
                },
                "object" : [
                {
                    "@" : {
                        "junos-configuration-metadata:comment" : "/* protect an object with an
identifier */",
                        "junos-configuration-metadata:protect" : (false | true)
                    },
                    "name" : "identifier",
                    "@statement-name" : {
                        "junos-configuration-metadata:comment" : "/* protect a statement */",
                        "junos-configuration-metadata:protect" : (false | true)
                    }
                }
                ]
            }
        /* closing braces for parent hierarchies */
    }
}
</configuration-json>
```

For example, the following RPC protects the `[edit protocols isis]` hierarchy level, the `apply-groups` leaf-list statement, and the `host-name` leaf statement, and it removes the protect attribute for the specified event policy.

```
<rpc>
<load-configuration format="json">
<configuration-json>
{
    "configuration" : {
        "@apply-groups" : {
            "junos-configuration-metadata:protect" : true
        },
        "system" : {
            "@host-name" : {
                "junos-configuration-metadata:protect" : true
            }

        },
        "event-options" : {
            "policy" : [
            {
                "@" : {
                    "junos-configuration-metadata:protect" : false
                },
                "name" : "raise-trap-on-ospf-nbrdown"
            }
            ]
        },
        "protocols" : {
            "isis" : {
                "@" : {
                    "junos-configuration-metadata:protect" : true
                }
            }
        }
    }
}
</configuration-json>
</load-configuration>
</rpc>
```

XML

To protect or unprotect a configuration object, include the `jcmd:protect="true"` or `jcmd:protect="false"` annotation, respectively, as an XML attribute in the opening tag of that configuration element.

The following RPC protects the `[edit protocols isis]` hierarchy level, the `apply-groups` leaf-list statement, and the `host-name` leaf statement, and it removes the protect attribute for the specified event policy.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <apply-groups xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:protect="true"/>
        <system>
          <host-name xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:protect="true"/>
        </system>
        <protocols>
           <isis xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:protect="true"/>
        </protocols>
        <event-options>
          <policy xmlns:jcmd="http://yang.juniper.net/junos/jcmd" jcmd:protect="false">
            <name>raise-trap-on-ospf-nbrdown</name>
          </policy>
        </event-options>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

`openconfig-metadata` Module Overview

The `openconfig-metadata` YANG module includes metadata annotations defined by the OpenConfig working group. The module defines the `protobuf-metadata` annotation, which enables you to store metadata about the configuration directly within the configuration for easy reference.

Junos devices support the `openconfig-metadata:protobuf-metadata` annotation with the following constraints:

- You can configure only one `protobuf-metadata` annotation and only at the root level of the configuration hierarchy.
- You can only configure and view the annotation in JSON configuration data.
- The annotation is of type binary, but you must encode the binary value in the base64 encoding scheme before loading the annotation on the device.

Junos devices support configuring the `openconfig-metadata:protobuf-metadata` annotation by default. However, to enable the device to emit the capability in the NETCONF capabilities exchange and emit the annotation in the configuration data, you must configure the device as follows:

1. Require the NETCONF server to advertise standard YANG modules, such as OpenConfig modules, in the capabilities exchange.

   ```
   [edit]
   user@host# set system services netconf hello-message yang-module-capabilities advertise-
   standard-yang-modules
   ```

2. Configure the device to enforce YANG-compliant NETCONF sessions.

   ```
   [edit]
   user@host# set system services netconf yang-compliant
   ```

3. (Optional) Unhide the OpenConfig schema, if you intend to view OpenConfig statements, including the annotation, in the CLI.

   ```
   [edit]
   user@host# set system schema openconfig unhide
   ```

4. Commit the configuration.

```
[edit]
user@host# commit
```

After you configure the device to advertise standard YANG modules in the NETCONF capabilities exchange, devices that support `openconfig-metadata` annotations advertise the following capability in the `hello` message:

```
<capability>http://openconfig.net/yang/openconfig-metadata?module=openconfig-
metadata&amp;revision=2020-08-06</capability>
```

You use the gNMI `set()` operation to load the `openconfig-metadata:protobuf-metadata` annotation as part of your JSON configuration data.

```
{
    "configuration" : {
        "@" : {
            "openconfig-metadata:protobuf-metadata": "dGhpcyBpcyB0ZXN0IGRhdGEK"   // base64
encoded string per RFC 7951 encoding rules.
        },
        // configuration statements
    }
}
```

When you request JSON configuration data, as described in "View Metadata Annotations in Configuration Data" on page 518, the output displays the OpenConfig configuration, including the annotation, after the Junos configuration data. For example:

```
<rpc><get-configuration format="json"/></rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.juniper.net/
junos/22.3R1/junos">
{
    "configuration" : {
        "@" : {
            "xmlns" : "http://xml.juniper.net/xnm/1.1/xnm",
            "junos:changed-seconds" : "1658526284",
            "junos:changed-localtime" : "2022-07-22 14:44:44 PDT"
        },
        "version" : "22.3R1-EVO",
        ...
    },
    "@" : {
        "openconfig-metadata:protobuf-metadata" : "dGhpcyBpcyB0ZXN0IGRhdGEK"
    },
    "openconfig-interfaces:interfaces" : {
        "interface" : [
        {
            "name" : "et-1/0/1",
            "config" : {
                "type" : "IF_ETHERNET",
                "description" : "CE1"
            }
        }
        ]
    }
}
</rpc-reply>
```

View Metadata Annotations in Configuration Data

The Junos device emits YANG metadata annotations in the Junos configuration within YANG-compliant NETCONF sessions. When you configure NETCONF sessions to be YANG-compliant and retrieve the configuration using the `<get-config/>` or `<get-configuration/>` RPC, the device encodes the annotations as per RFC 7952, *Defining and Using Metadata with YANG*.

To view the configuration with the YANG annotations encoded as per RFC 7952:

1. Configure the device to enforce YANG-compliant NETCONF sessions.

```
[edit]
user@host# set system services netconf yang-compliant
user@host# commit
```

2. Retrieve the configuration using the `<get-config>` or `<get-configuration>` RPC.

   - Use the NETCONF `<get-config>` operation to retrieve XML configuration data.

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
```

- Use the Junos XML protocol `<get-configuration>` operation to retrieve JSON or XML configuration data.

```
<rpc><get-configuration format="json"/></rpc>

<rpc><get-configuration format="xml"/></rpc>
```

NOTE: Junos devices only support the `openconfig-metadata:protobuf-metadata` annotation for JSON encoding. Thus, you can only use the gNMI `get()` operation or the Junos XML protocol `<get-configuration format="json">` RPC to view the annotation in JSON configuration data.

Use Juniper Networks YANG Modules

SUMMARY

Learn how to obtain Juniper Networks YANG modules and how to import a module into another module.

IN THIS SECTION

Juniper Networks publishes YANG modules that define the configuration hierarchies, RPCs, state data, and YANG extensions for Junos devices. This topic details how to download the YANG modules or generate them on a device as well as how to import a module into another module.

Obtain Juniper Networks YANG Data Models

You can retrieve the Juniper Networks YANG data models by:

- Downloading the modules from the Juniper Networks website at https://www.juniper.net/support/downloads
- Downloading the modules from the Juniper/yang GitHub repository
- Generating the modules on a Junos device either through a local or remote session

Junos OS YANG modules are specific to a device family. For a given OS and release, the YANG modules download file and GitHub repository include a separate directory for each device family as well as a **common** directory. Each family-specific directory contains the modules that are supported on the platforms in that family. The **common** directory contains the modules that are common to all device families. For more information about the device families, see "Understanding Junos YANG Modules" on page 471.

When you generate YANG modules on a local device, the modules include both native Junos modules as well as any standard or custom modules that have been added to the device. The native YANG modules generated on a local device, by default, contain family-specific schemas. The family-specific schemas are identical across all devices in the given device family.

Starting in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1, we provide all YANG data models for a given OS and release in a single download package and GitHub repository folder. The package and repository include:

- Native configuration, state, and RPC data models
- OpenConfig configuration and state models supported by that OS
- IETF models supported by that OS

Download YANG Modules from Juniper Networks

To download the YANG modules from the Juniper Networks site:

1. Access the downloads page at https://support.juniper.net/support/downloads.
2. Select your product.
3. In the drop-down menus, select the appropriate OS and version.
4. Expand the **Tools** section to display the available downloads.
5. In the **Downloads** column, click the link for the YANG modules package you want to download.

6. Review and accept the End User License Agreement.
7. Follow the instructions on the download page to download the file.

NOTE: If your particular product page does not have the YANG modules available for download, you can download the modules from the Juniper/yang GitHub repository or generate the modules on the Junos device.

Generate YANG Modules on Junos Devices

To generate the YANG modules from the Junos OS or Junos OS Evolved CLI:

1. Log in to the Junos device.
2. (Optional) By default, Junos devices emit YANG modules with family-specific schemas. To generate modules with device-specific schemas instead, configure the `device-specific` statement.

   ```
   [edit system services netconf]
   user@host# set yang-modules device-specific
   user@host# commit and-quit
   ```

3. Create the directory where the device will store the output files, if it does not exist.

   ```
   user@host> file make-directory /var/tmp/yang
   ```

4. (Optional) To see a list of available Junos YANG modules, invoke the context-sensitive help.

   ```
   user@host> show system schema module ?
   ```

5. Execute the `show system schema` operational mode command. Specify which modules to generate and the directory for storing the output files.

   ```
   user@host> show system schema format yang module module-name output-directory file-path
   ```

   For example:

   ```
   user@host> show system schema format yang module all output-directory /var/tmp/yang
   ```

The device generates the YANG modules in the specified output directory.

Generate YANG Modules from a Remote Session

To generate the YANG modules from a remote session:

1. Connect to the Junos device. For example:

```
user@server$ ssh user@host.example.net -p 830 -s netconf
```

2. Execute the `<get-yang-schema>` RPC. Specify the module or collection name and the output directory.

```
<rpc>
   <get-yang-schema>
      <format>yang</format>
      <identifier>all-rpc</identifier>
      <output-directory>/var/home/user</output-directory>
   </get-yang-schema>
</rpc>
```

The device generates the YANG modules in the specified output directory.

TIP: You can also use the `<get-schema>` Network Configuration Protocol (NETCONF) operation to retrieve a YANG module from the device. For additional information, see "Retrieve Schemas" on page 159.

Importing Juniper Networks YANG Modules

You can use YANG-based tools to leverage the Juniper Networks YANG modules. If you are developing custom YANG modules, you can reference definitions in the Juniper Networks YANG modules by importing the modules into your custom module.

To import a Juniper Networks YANG module into an existing module:

1. Include the import statement, specify the module name, and assign the prefix to use with the definitions from the imported module.

```
module acme-system {
   namespace "http://acme.example.com/system";
   prefix "acme";

   import junos-conf-root {
      prefix "jc";
```

```
    }
    import junos-extension {
       prefix "junos";
    }
 …
 }
```

2. Reference definitions in the module by using the locally defined prefix, a colon, and the node identifier or keyword.

Platform-Specific YANG Module Behavior

Use Feature Explorer to confirm platform and release support for specific features.

Use the following table to review platform-specific behaviors for your platform:

Table 29: Platform-Specific Behavior

Platform	Difference
QFX Series	• In Junos OS Evolved Release 23.4R2 and later, native YANG modules use the `junos` device family identifier instead of `junos-qfx`. To emit device-specific schemas that use the `junos-qfx` family identifier instead, configure the `device-specific` and `emit-family-ns-and-module-name` statements at the `[edit system services netconf yang-modules]` hierarchy level.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
23.4R1 and 23.4R1-EVO	Starting in Junos OS Release 23.4R1 and Junos OS Evolved Release 23.4R1, we provide all YANG data models for a given OS and release in a single download package and GitHub repository folder.
23.4R1-EVO	Starting in Junos OS Evolved Release 23.4R1, we publish the Junos OS Evolved native YANG modules on the Juniper Networks download site and on GitHub. In earlier releases, you must generate the modules on the device.

19.1R2 and 19.2R1	Starting in Junos OS Release 19.1R2 and 19.2R1, the `show system schema` command and `get-yang-schema` RPC must include the `output-directory` option to specify the directory in which to generate the output files. In earlier releases, you can omit the `output-directory` option when requesting a single module to display the module in standard output.

RELATED DOCUMENTATION

Understanding YANG on Devices Running Junos OS | **470**

Understanding Junos YANG Modules | **471**

Understanding the Junos DDL Extensions YANG Module | **500**

Understanding the YANG Modules for Junos Operational Commands | **484**

show system schema

CHAPTER 18

Create and Use Non-Native YANG Modules

IN THIS CHAPTER

Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS

YANG is a standards-based, extensible data modeling language that is used to model the configuration and operational state data, remote procedure calls (RPCs), and server event notifications of network devices. You can load standard or custom YANG models onto Junos devices to add data models that are not natively supported by Junos OS. Adding YANG models enables you to create device-agnostic and vendor-neutral operational and configuration models for managing devices from multiple vendors.

When you add custom YANG data models to Junos devices, you must supply a script that handles the translation logic between the YANG data model and Junos OS on that device. There are two types of scripts:

- *Action scripts* are Stylesheet Language Alternative SyntaX (SLAX) or Python scripts that act as handlers for your custom YANG RPCs. The YANG RPC definition uses a Junos OS YANG extension to reference the appropriate action script, which is invoked when you execute the RPC.

- *Translation scripts* are SLAX or Python scripts that map your custom configuration syntax (defined by your YANG model) to Junos OS syntax and then load the translated data into the configuration as a transient change during the commit operation. When you load and commit configuration data in the nonnative hierarchies, Junos OS invokes the script to perform the translation and emit the transient change.

To use custom YANG data models on Junos devices, you must add the YANG modules and associated scripts to the device by issuing the `request system yang add` command. Junos OS validates the syntax of the modules and scripts, rebuilds its schema to include the new data models, and then validates the active configuration against this schema. The device validates the modules and scripts as you add them. However, we recommend that you use the `request system yang validate` command first to validate the syntax before adding the modules.

NOTE: In multichassis systems, you must download and add the modules and scripts to each node in the system.

NOTE: To install OpenConfig modules that are packaged as a compressed tar file, use the `request system software add` command.

When you add YANG modules and scripts to Junos devices, you must associate them with a package. Packages have a unique identifier and represent a collection of related modules, translation scripts, and action scripts. You reference the package identifier if you later update modules and scripts in that package, enable or disable translation scripts associated with the package, or delete that group of modules and scripts from the device.

When you add, update, or remove YANG modules and scripts on the device by issuing the appropriate operational commands, you do not need to reboot the device in order for the changes to take effect. Newly added RPCs and configuration hierarchies are immediately available for use, and installed translation scripts are enabled by default. You can disable translation scripts in a package at any time without removing the package and associated files from the device, which can be useful for troubleshooting translation issues. When you disable translation for a package, you can still configure and commit the statements and hierarchies added by the YANG modules in that package. However, the device does not translate and commit the corresponding Junos OS configuration as a transient configuration change during the commit operation.

Before installing software on a device that has one or more custom YANG data models added to it, you must remove all configuration data corresponding to the custom YANG data models from the active configuration. After the software installation is complete, add the YANG packages and corresponding configuration data back to the device, if appropriate. For more information see "Managing YANG Packages and Configurations During a Software Upgrade or Downgrade" on page 533.

RELATED DOCUMENTATION

Manage YANG Packages, Modules, and Scripts on Junos Devices | **527**

Disable and Enable YANG Translation Scripts on Devices Running Junos OS | **541**

Create Translation Scripts for YANG Configuration Models | **536**

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Manage YANG Packages, Modules, and Scripts on Junos Devices

SUMMARY

Load custom YANG packages on Junos devices to add your own remote procedure calls (RPCs) and data models to the device.

IN THIS SECTION

You can load custom YANG modules on Junos devices to add RPCs and data models that the device does not support natively but can support through translation. When you load nonnative YANG data models onto the device, you must also load any translation scripts, action scripts, and deviation modules required by those data models.

Junos devices use packages to identify a collection of related YANG modules, translation scripts, and action scripts. Each package has a unique identifier. When you add YANG modules and scripts to the device, you must associate them with a new or existing package. This topic discusses how to create, update, or delete YANG packages and add or update their associated modules and scripts.

Before you add, update, or delete YANG packages on a device, you should understand the following points for working with YANG packages:

- To prevent CLI-related errors or configuration database errors, we recommend that you do not perform any CLI operations, change the configuration, or terminate the operation while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.
- You cannot use the `run` command in configuration mode to add, delete, or update YANG packages.
- When you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules.
- After you add, update, or delete YANG packages, the device rebuilds the schema. Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.
- Junos OS does not support using `configure private` mode to configure statements corresponding to third-party YANG data models, for example, OpenConfig or custom YANG data models.

Create a YANG Package and Add Modules and Scripts

To validate YANG modules and scripts and add them to a new package:

1. Download the YANG modules and any necessary scripts to any directory on the device.
2. Ensure that any unsigned Python action scripts are owned by either root or a user in the Junos OS `super-user` login class and that only the file owner has write permission for the file.

   NOTE: Users can only execute unsigned Python scripts on Junos devices when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

3. (Optional) Validate the syntax of the modules and scripts.

   ```
   user@host> request system yang validate action-script [scripts] module [modules] translation-
   script [scripts]
   ```

4. Create a YANG package with a unique identifier, and specify the file paths for the modules and scripts that are part of that package. Additionally, include any deviation modules that identify deviations for the modules in that package.

   ```
   user@host> request system yang add package package-name module [modules] deviation-
   module [modules] translation-script [scripts] action-script [scripts]
   ```

NOTE: You can specify the absolute or relative path to a single file, or you can add multiple files by specifying a space-delimited list of file paths enclosed in brackets.

NOTE: To install OpenConfig modules that are packaged as a compressed tar file, use the `request system software add` command. OpenConfig modules and scripts that are installed by issuing the `request system software add` command are always associated with the package identifier `openconfig`.

5. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

6. Verify that the device created the package and that it contains the correct modules and scripts.

```
user@host> show system yang package package-name
Package ID            :package-name
YANG Module(s)        :modules
Action Script(s)      :action scripts
Translation Script(s) :translation scripts
Translation script status is enabled
```

7. If the package includes Python translation scripts or Python action scripts, enable the device to execute unsigned Python scripts. Configure the `language python` or `language python3` statement, as appropriate for the Junos OS release.

```
[edit]
user@host# set system scripts language (python | python3)
user@host# commit
```

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

8. On multichassis systems, repeat steps 1 through 7 on each node in the system.

When you create a package, the device stores copies of the module and script files in a new location. The device also stores copies of the action script and translation script files under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories, respectively. After the modules and scripts are validated and added to the device, Junos OS rebuilds its schema to include the new data models. The device then validates the active configuration against this schema. Newly added RPCs and configuration hierarchies are immediately available for use.

Update a YANG Package with New or Modified Modules and Scripts

You create a YANG package by executing the `request system yang add` command. To update an existing package to either add new modules and scripts to the package or update existing modules and scripts in the package, you must use the `request system yang update` command.

To update a YANG package with new or modified modules and scripts:

1. Download the modules and scripts to any directory on the device.
2. Ensure that any unsigned Python action scripts are owned by either root or a user in the Junos OS `super-user` login class and that only the file owner has write permission for the file.

   NOTE: Users can only execute unsigned Python scripts on Junos devices when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

3. (Optional) Validate the syntax of the modules and scripts.

```
user@host> request system yang validate action-script [scripts] module [modules] translation-
script [scripts]
```

4. Update the YANG package by issuing the `request system yang update` command, and specify the file paths for the new and modified modules and scripts.

```
user@host> request system yang update package-name module [modules] deviation-
module [modules] translation-script [scripts] action-script [scripts]
```

NOTE: You can specify the absolute or relative path to a single file, or you can update multiple files by specifying a space-delimited list of file paths enclosed in brackets.

5. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

6. If the package includes Python translation scripts or Python action scripts, enable the device to execute unsigned Python scripts. Configure the `language python` or `language python3` statement, as appropriate for the Junos OS release, if it is not already configured.

```
[edit]
user@host# set system scripts language (python | python3)
user@host# commit
```

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

7. On multichassis systems, repeat steps 1 through 6 on each node in the system.

When you update a package, the device stores copies of the new and modified module and script files. Junos OS rebuilds its schema to include any changes to the data models associated with that package. The device then validates the active configuration against this schema.

Delete a YANG Package

CAUTION: Before you delete a YANG package from a Junos device, ensure that the active configuration does not contain configuration data that has dependencies on the data models added by that package.

To delete a YANG package and all modules and scripts associated with that package from a Junos device:

1. Review the active configuration to determine if there are any dependencies on the YANG modules that will be deleted.
2. If the configuration contains dependencies on the modules, update the configuration to remove the dependencies.
3. Delete the package and associated modules and scripts by issuing the `request system yang delete` command with the appropriate package identifier.

```
user@host> request system yang delete package-name
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...
```

NOTE: You must use the `request system software delete` command to remove OpenConfig packages that were installed from a compressed tar file by issuing the `request system software add` command.

4. If the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`.

```
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...

WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

When you delete a package, Junos OS rebuilds its schema to remove the data models associated with that package. The device then validates the active configuration against this schema. The device removes the copies of the module and script files that it generated when you created the package. The device also removes the copies of the package's action script and translation script files that reside under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories. If you downloaded the original module and script files to a different location, the original files remain unchanged.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
22.3R1-EVO	Starting in Junos OS Evolved Release 22.3R1, Junos OS Evolved uses Python 3 to execute YANG action and translation scripts.
20.2R1	Starting in Junos OS Release 20.2R1, Junos OS uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS uses Python 2.7 to execute these scripts.
18.3R1	Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the `run` command is not supported.

RELATED DOCUMENTATION

Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS | 525

Managing YANG Packages and Configurations During a Software Upgrade or Downgrade | 533

request system yang add

request system yang delete

request system yang update

show system yang package

Managing YANG Packages and Configurations During a Software Upgrade or Downgrade

IN THIS SECTION

Certain devices running Junos OS enable you to load custom YANG modules on the device to add data models that are not natively supported by Junos OS. When you add, update, or delete a YANG data model, Junos OS rebuilds its schema and then validates the active configuration against the updated schema.

When you upgrade or downgrade Junos OS, by default, the system validates the software package or bundle against the current configuration. During the installation, the schema for custom YANG data models is not available. As a result, if the active configuration contains dependencies on these models, the software validation fails, which causes the upgrade or downgrade to fail.

In addition, devices that are running Junos OS based on FreeBSD version 6 remove custom YANG packages from the device during the software installation process. For this Junos OS variant, if the active configuration contains dependencies on custom YANG data models, the software installation fails even if you do not validate the software against the configuration, because the configuration data cannot be validated during the initial boot-time commit.

For these reasons, before you upgrade or downgrade the Junos OS image on a device that has one or more custom YANG modules added to it, you must remove all configuration data corresponding to the custom YANG data models from the active configuration. After the software installation is complete, add the YANG packages and corresponding configuration data back to the device, if appropriate. The tasks are outlined in this topic.

NOTE: You do not need to delete configuration data corresponding to OpenConfig packages before upgrading or downgrading Junos OS.

Backing up and Deleting the Configuration Data

If the configuration contains dependencies on custom YANG data models:

1. If you plan to restore the configuration data that corresponds to the nonnative YANG data models after the software is updated, save a copy of either the entire configuration or the configuration data corresponding to the YANG data models, as appropriate.

   - To save the entire configuration:

   ```
   user@host> show configuration | save (filename | url)
   ```

   - To save configuration data under a specific hierarchy level:

   ```
   user@host> show configuration path-to-yang-statement-hierarchy | save (filename | url)
   ```

2. In configuration mode, delete the portions of the configuration that depend on the custom YANG data models.

```
[edit]
user@host# delete path-to-yang-statement-hierarchy
```

3. Commit the changes.

```
[edit]
user@host# commit
```

4. Prior to performing the software installation, ensure that the saved configuration data and the YANG module and script files are saved to a local or remote location that will preserve the files during the installation and that will be accessible after the installation is complete.

Restoring the YANG Packages and Configuration Data

After the software installation is complete, load the YANG packages onto the device (where required), and restore the configuration data associated with the packages, if appropriate. During a software upgrade or downgrade, devices running Junos OS with upgraded FreeBSD preserve custom YANG packages, whereas devices running Junos OS based on FreeBSD version 6 delete the packages.

1. Load the YANG packages (devices running Junos OS based on FreeBSD version 6 only).

```
user@host> request system yang add package package-name module [modules] deviation-
module [modules] translation-script [scripts] action-script [scripts]
```

2. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations, change the configuration, or terminate the operation while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

3. In configuration mode, load the configuration data associated with the YANG packages.

   For example, to load the configuration data from a file relative to the top level of the configuration statement hierarchy:

```
[edit]
user@host# load merge (filename | url)
```

NOTE: For more information about loading configuration data, see the *CLI User Guide*.

4. Commit the changes.

```
[edit]
user@host# commit
```

RELATED DOCUMENTATION

Manage YANG Packages, Modules, and Scripts on Junos Devices | **527**

Create Translation Scripts for YANG Configuration Models

You can load YANG modules on Junos devices to add data models that are not natively supported by the OS but can be supported by translation. When you extend the configuration hierarchy with nonnative YANG data models, you must also supply one or more translation scripts that provide the logic to map the nonnative configuration syntax to the corresponding Junos OS syntax.

Translation scripts perform two main functions:

- Convert the configuration data corresponding to the nonnative YANG data models into Junos OS syntax

- Add the translated configuration data as a transient change in the checkout configuration during the commit operation

Translation scripts can be written in either Python or SLAX and are similar to commit scripts in structure. For information about creating SLAX and Python scripts that generate transient changes in the configuration, see the Automation Scripting User Guide.

You use the `request system yang add` or `request system yang update` commands to add YANG modules and their associated translation scripts to a new or existing YANG package on the device. After you add the modules and translation scripts to the device, you can configure the statements and hierarchies in the data model added by those modules. When you load and commit the configuration data, the device calls the script to perform the translation and generate the transient configuration change.

This topic discusses the general structure for translation scripts. The specific translation logic required in the actual script depends on the custom hierarchies added to the schema and is beyond the scope of this topic.

To create the framework for translation scripts that are used on Junos devices:

1. In your favorite editor, create a new file that uses the **.slax** or **.py** file extension, as appropriate.
2. Include the necessary boilerplate required for that script's language, which is identical to the boilerplate for commit scripts, and also include any required namespace declarations for your data models.

   - SLAX code:

   ```
   version 1.0;
   ns junos = "http://xml.juniper.net/junos/*/junos";
   ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
   ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
   ns prefix = "namespace";
   import "../import/junos.xsl";

   match configuration {
      /*
       * insert your code here
       */
   }
   ```

   - Python code:

   ```
   from junos import Junos_Context
   from junos import Junos_Configuration
   ```

```
import jcs

if __name__ == '__main__':
   /*
    * insert your code here
    */
```

NOTE: Translation scripts must fully qualify identifiers for nonnative YANG data models in the translation code.

NOTE: For information about commit script boilerplate code, see *Required Boilerplate for Commit Scripts* and the Automation Scripting User Guide.

3. Add code that maps the nonnative configuration data into the equivalent Junos OS syntax and stores the translated configuration data in a variable.

   - SLAX sample code:

```
match configuration {

    /* translation code */

    var $final = {
       /*
        * translated configuration
        */
    }
}
```

   - Python sample code:

```
if __name__ == '__main__':

    /* translation code */

    final = """
           /*
            * Junos XML elements representing translated configuration
            */
    """
```

4. Add the translated content to the checkout configuration as a transient configuration change by calling the `jcs:emit-change()` template in SLAX scripts or the `jcs.emit_change()` function in Python scripts with the translated configuration and `transient-change` tag as arguments.

   - SLAX sample code:

```
match configuration {

    /* translation code */

    var $final = {
       /*
        * translated configuration
        */
    }
    call jcs:emit-change($content=$final, $tag='transient-change');
}
```

   - Python sample code:

```
if __name__ == '__main__':

    /* translation code */

    final = """
           /*
            * Junos XML elements representing translated configuration
            */
    """
    jcs.emit_change(final, "transient-change", "xml")
```

NOTE: In SLAX scripts, you can also generate the transient change by emitting the translated configuration inside of a `<transient-change>` element instead of calling the `jcs:emit-change()` template.

On the device, perform the following tasks before adding the translation script to a YANG package:

1. If the translation script is written in Python, enable the device to execute unsigned Python scripts by configuring the `language python` or `language python3` statement, as appropriate for the Junos OS release.

   ```
   [edit]
   user@host# set system scripts language (python | python3)
   ```

   NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

2. Download the script to the device, and optionally validate the syntax.

   ```
   user@host> request system yang validate translation-script script
   ```

Before you can use translation scripts on a device, you must add the scripts and associated modules to a new or existing YANG package by issuing the `request system yang add` or `request system yang update` command. After the modules and scripts are added, the translation scripts are automatically invoked when you commit configuration data in the corresponding data models.

When you configure statements that correspond to third-party YANG data models, for example, OpenConfig or custom YANG data models, the following features are *not* supported:

- Using `configure batch` or `configure private` mode
- Configuring statements under the `[edit groups]` hierarchy

The active and candidate configurations contain the configuration data for the nonnative YANG data models in the syntax defined by those models. However, because the translated configuration data is committed as a transient change, the active and candidate configurations do not explicitly display the translated data in the Junos OS syntax when you issue the `show` or `show configuration` commands. To apply YANG translation scripts when you view the configuration, use the `| display translation-scripts` filter.

To view the complete post-inheritance configuration with the translated data (transient changes) explicitly included, append the `| display translation-scripts` filter to the `show configuration` command in operational mode or the `show` command in configuration mode. To view just the nonnative configuration data after translation, use the `| display translation-scripts translated-config` filter.

In configuration mode, to display just the changes to the configuration data corresponding to nonnative YANG data models before or after translation scripts are applied, append the `configured-delta` or `translated-delta` keyword, respectively, to the `show | display translation-scripts` command. In both cases, the XML output displays the deleted configuration data, followed by the new configuration data.

For more information about the `| display translation-scripts` filter, see "Commit and Display Configuration Data for Nonnative YANG Modules" on page 543.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
22.3R1	Starting in Junos OS Evolved Release 22.3R1, Junos OS Evolved uses Python 3 to execute YANG action and translation scripts.
20.2R1	Starting in Junos OS Release 20.2R1, Junos OS uses Python 3 to execute YANG action and translation scripts.

RELATED DOCUMENTATION

Disable and Enable YANG Translation Scripts on Devices Running Junos OS | **541**

Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS | **525**

Disable and Enable YANG Translation Scripts on Devices Running Junos OS

You can load standard (IETF, OpenConfig) or custom YANG data models on devices running Junos OS to add data models that are not natively supported by Junos OS but can be supported by translation. When you extend the configuration hierarchy with nonnative data models, you must also supply one or more translation scripts. Tranlation scripts perform two main functions:

- Map the custom configuration syntax defined by the YANG data model to the corresponding Junos OS syntax
- Add the translated data to the checkout configuration as a transient change during the commit operation

When you add translation scripts to the device with a new or existing YANG package, they are enabled by default. You can disable the translation scripts in a YANG package at any time without removing the package and associated files from the device, which can be useful for troubleshooting translation issues. After you disable translation for a package and commit the configuration, the configuration data associated with the YANG data models in that package can be present in the active configuration, but the configuration has no impact on the functioning of the device.

When translation is disabled, you can still configure and commit the statements and hierarchies in the data models added by that package. However, the device does not commit the corresponding Junos OS configuration statements as transient changes during the commit operation for any statements in the data models added by that package, even for those statements that were committed prior to disabling translation.

To disable translation scripts for a given YANG package that is installed on a device running Junos OS:

1. Issue the `request system yang disable` command, and specify the package identifier.

```
user@host> request system yang disable package-name
```

2. Verify that the status of the translation scripts in the package is `disabled`.

```
user@host> show system yang package package-name
Package ID              :package-name
YANG Module(s)          :modules
Translation Script(s) :translation scripts
Translation script status is disabled
```

NOTE: When you disable translation for a package, the device retains any transient configuration changes that were committed prior to disabling translation until the next commit operation.

NOTE: In configuration mode, you can issue the `show | display translation-scripts translated-config` command to verify which configured statements from nonnative YANG data models will be translated and committed during a `commit` operation. The command output does not include (and the device does not commit) the corresponding Junos OS configuration for those data models for which translation has been disabled.

To enable translation scripts for a given YANG package that is installed on a device running Junos OS:

1. Issue the `request system yang enable` command, and provide the package identifier.

```
user@host> request system yang enable package-name
```

2. Verify that the status of the translation scripts in the package is `enabled`.

```
user@host> show system yang package package-name
Package ID            :package-name
YANG Module(s)        :modules
Translation Script(s) :translation scripts
Translation script status is enabled
```

RELATED DOCUMENTATION

Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS | 525

request system yang disable

request system yang enable

show system yang package

Commit and Display Configuration Data for Nonnative YANG Modules

You can load standardized or custom YANG modules on Junos devices to add data models that the device does not support natively but can support through translation. When you extend the configuration hierarchy with new data models, you must also supply one or more translation scripts. Translation scripts provide the translation logic to map the nonnative configuration syntax to Junos OS syntax. The device enables translation scripts as soon as you issue the `request system yang add` or `request system yang update` command to add them to the device.

You configure nonnative data models in the candidate configuration using the syntax defined for those models. When you configure statements that correspond to third-party YANG data models, for example, OpenConfig or custom YANG data models, the following features are *not* supported:

- Using `configure batch` or `configure private` mode
- Configuring statements under the `[edit groups]` hierarchy

When you commit the configuration, the translation scripts translate the data for those models and commit the corresponding Junos OS configuration as a transient change in the checkout configuration.

NOTE: XPath expression evaluations for the following YANG keywords are disabled by default during commit operations: `leafref`, `must`, and `when`.

The candidate and active configurations contain the configuration data for nonnative YANG data models in the syntax defined by those models. The translated configuration data is committed as a transient change. As a result, the candidate and active configurations do not explicitly display the translated data in the Junos OS syntax when you view the configuration by using commands such as `show` or `show configuration`.

You can explicitly display the translated data in Junos OS syntax in the candidate or active configuration. You append the `| display translation-scripts` filter to the `show` command in configuration mode or to the `show configuration` command in operational mode. Applying the filter displays the post-inheritance configuration with the translated configuration data from all enabled translation scripts included.

NOTE: You can only apply the `| display translation-scripts` filter to the complete Junos OS configuration. You cannot filter subsections of the configuration hierarchy.

In operational mode, issue the following command to view the committed configuration with translation scripts applied:

```
user@host> show configuration | display translation-scripts
```

Similarly, in configuration mode, issue the following command to view the candidate configuration with translation scripts applied. You must be at the top of the configuration hierarchy to use the filter.

```
[edit]
user@host# show | display translation-scripts
```

The output, which is truncated in this example, displays the complete post-inheritance configuration and includes the nonnative configuration data as well as the translation of that data.

```
  ## Last changed: 2025-09-13 16:37:42 PDT
  version 25.2R1.9;
  system {
      host-name host;
      domain-name example.com;
      ...
      /* Translated data */
      scripts {
          op {
            file test.slax;
          }
      }
```

```
        ...
    }
    ...
    /* Nonnative configuration data */
    myconfig:myscript {
        op {
            filename test.slax;
        }
    }
```

Alternatively, you can view just the translated portions of the hierarchy corresponding to nonnative YANG data models by appending the `translated-config` keyword to the `| display translation-scripts` filter. In operational mode, the `translated-config` keyword returns the translated data for nonnative YANG data models present in the committed configuration. In configuration mode, the `translated-config` keyword returns the translated data for nonnative YANG data models present in the candidate configuration, which includes both committed and uncommitted configuration data.

```
user@host> show | display translation-scripts translated-config
```

```
    system {
        scripts {
            op {
                file test.slax;
            }
        }
    }
```

The candidate configuration reflects the configuration data that has been configured, but not necessarily committed, on the device. In configuration mode, you can display just the configuration differences in the hierarchies corresponding to nonnative YANG data models before or after translation scripts are applied. To display the differences, append the `configured-delta` or `translated-delta` keyword to the `show | display translation-scripts` command. In both cases, the XML output displays the deleted configuration data, followed by the new configuration data.

For example, to view the uncommitted configuration changes for the nonnative data models in the syntax defined by those data models, use the `show | display translation-scripts configured-delta` configuration mode command.

```
[edit]
user@host# show | display translation-scripts configured-delta
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/25.2R1.9/junos">
    <configuration operation="delete">
    </configuration>
    <configuration operation="create">
        <myscript xmlns="http://jnpr.net/yang/myscript" operation="create">
            <op>
                <filename>test2.slax</filename>
            </op>
        </myscript>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

To view the uncommitted configuration changes for the nonnative data models after translation into Junos OS syntax, use the `show | display translation-scripts translated-delta` configuration mode command. For example:

```
[edit]
user@host# show | display translation-scripts translated-delta
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/R1/junos">
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos">
        <system>
              <scripts>
                <op>
                  <file>
                    <name>test2.slax</name>
                  </file>
                </op>
              </scripts>
        </system>
    </configuration>
    <!-- EOF -->
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

In configuration mode, you can use the various filters to better understand which transient changes the device will commit for nonnative data models. To verify all Junos OS statements that will be committed as transient changes by translation scripts during the `commit` operation, issue the `show | display translation-`

`scripts translated-config` command before committing the candidate configuration. To verify the Junos OS statements that will be committed for just the changed configuration data, issue the `show | display translation-scripts translated-delta` command. If you disable translation scripts for a package, the output for these commands does not include (and the device does not commit) the corresponding Junos OS configuration for those data models in the package.

NOTE: The presence of nonnative configuration data in the active configuration does not guarantee that the corresponding translated configuration is present as a transient change. If you disable translation and then commit nonnative configuration data, the nonnative data is present in the committed configuration. However, the device does not commit the corresponding Junos OS configuration statements as transient changes for any statements in the data models added by that package. This even includes statements you already committed prior to disabling translation.

Table 30 on page 547 summarizes the different filters you can apply to the committed and candidate configurations to view configuration data corresponding to nonnative YANG data models. The table indicates the CLI mode for each filter, and the scope and syntax of the output. By selecting different filters, you can view the entire configuration, the translated portions of the configuration, or the uncommitted configuration changes. You can also view the configuration data both before and after processing by translation scripts. In configuration mode, this enables you to better determine the Junos OS changes that the device will commit for the nonnative hierarchies.

Table 30: `| display translation-scripts` Command

Filter	Mode	Description	Syntax and Format of Output
`| display translation-scripts`	Operational	Return the complete, post-inheritance committed configuration and include the translation of the nonnative data into Junos OS syntax.	YANG data model and Junos OS syntax as ASCII text

Table 30: `| display translation-scripts` Command *(Continued)*

Filter	Mode	Description	Syntax and Format of Output
	Configuration	Return the complete, post-inheritance candidate configuration and include the translation of the nonnative data into Junos OS syntax.	YANG data model and Junos OS syntax as ASCII text
`| display translation-scripts translated-config`	Operational	Return the translated data corresponding to all nonnative YANG data models in the committed configuration.	Junos OS ASCII text
	Configuration	Return the translated data corresponding to all nonnative YANG data models in the candidate configuration.	Junos OS ASCII text
`| display translation-scripts configured-delta`	Configuration	Return the uncommitted changes in the candidate configuration corresponding to nonnative YANG data models in the syntax defined by that model.	YANG data model XML
`| display translation-scripts translated-delta`	Configuration	Return the uncommitted changes in the candidate configuration corresponding to nonnative YANG data models after translation into Junos OS syntax.	Junos OS XML

Create Custom RPCs in YANG for Devices Running Junos OS

SUMMARY

You can also create YANG data models that define custom RPCs for devices running Junos OS or devices running Junos OS Evolved.

Juniper Networks provides YANG modules that define the remote procedure calls (RPCs) for operational commands on devices running Junos OS and devices running Junos OS Evolved. You can also create YANG data models that define custom RPCs for supported devices. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices. When you extend the operational command hierarchy with a custom YANG RPC, you must also supply an action script that serves as the handler for the RPC. The RPC definition references the action script; when you execute the RPC, the device invokes the script.

This topic outlines the general steps for creating a YANG module that defines a custom RPC for devices running Junos OS or devices running Junos OS Evolved. For information about creating an RPC action script and customizing the RPC's CLI output see "Create Action Scripts for YANG RPCs on Junos Devices" on page 556 and "Understanding Junos OS YANG Extensions for Formatting RPC Output" on page 590.

This section presents a generic template for a YANG module that defines an RPC for Junos devices. The template is followed by a detailed explanation of the different sections and statements in the template.

```
module module-name {
   namespace "namespace";
   prefix prefix;

   import junos-extension {
      prefix junos;
   }
   import junos-extension-odl {
      prefix junos-odl;
   }

   organization
      "organization";
   description
      "module-description";
```

```
    rpc rpc-name {
      description "RPC-description";

      junos:command "cli-command" {
        junos:action-execute {
          junos:script "action-script-filename";
        }
      }

      input {
        leaf input-param1 {
          type type;
          description description;
        }
        leaf input-param2 {
          type type;
          description description;
        }
        // additional leaf definitions
      }
      output {
        container output-container-name {

          container container-name {
            leaf output-param1 {
              type type;
              description description;
              // optional formatting statements
            }
            // additional leaf definitions

            junos-odl:format container-name-format {
              // CLI formatting for the parent container
            }
          }

          // Additional containers
        }
      }
    }
  }
```

You define RPCs within modules. The module name should be descriptive and indicate the general purpose of the RPCs defined in that module. The module namespace must be unique.

```
module module-name {
   namespace "namespace";
   prefix prefix;

}
```

NOTE: As per RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, the module name and the base name of the file in which the module resides must be identical. For example, if the module name is `get-if-info`, the module's filename must be **get-if-info.yang**.

The module must import the Junos OS DDL extensions module and define a prefix. The extensions module includes YANG extensions that are required in the definition of RPCs executed on devices running Junos OS or devices running Junos OS Evolved.

```
   import junos-extension {
      prefix junos;
   }
```

If any of the RPCs in the module render formatted ASCII output, the module must import the Junos OS ODL extensions module and define a prefix. The ODL extensions module defines YANG extensions that you use to precisely specify how to render text output. The device emits text output when you execute the operational command for that RPC in the CLI or when you request the RPC output in text format.

```
   import junos-extension-odl {
      prefix junos-odl;
   }
```

Include the organization responsible for the module as well as a description of the module.

```
   organization
      "organization";
   description
      "module-description";
```

Within the module, you can define one or more RPCs, each with a unique name. The RPC name is used to remotely execute the RPC, and thus should clearly indicate the RPC's purpose. The RPC purpose can be further clarified in the `description` statement. If you also define a CLI command for the RPC, the CLI displays the RPC description in the context-sensitive help for that command listing.

```
rpc rpc-name {
   description "RPC-description";

}
```

Within the RPC definition, define the `command`, `action-execute`, and `script` statements, which are Junos OS DDL extension statements. The `command` statement defines the operational command that you use to execute the RPC in the Junos OS CLI. To execute the RPC remotely, use the RPC name for the request tag.

Every RPC must define the `action-execute` statement and `script` substatement. The `script` substatement defines the name of the action script that is invoked when you execute the RPC. You must define one and only one action script for each RPC.

```
   junos:command "cli-command" {
      junos:action-execute {
         junos:script "action-script-filename";
      }
   }
```

NOTE: You must add the YANG module and action script to the device as part of a new or existing YANG package. Thus, you only need to provide the name and not the path of the action script for the `junos:script` statement.

NOTE: If your action script is written in Python, you must enable the device to execute unsigned Python scripts by configuring the `language python` or `language python3` statement under the `[edit system scripts]` hierarchy level on each device where you will execute the script.

Input parameters to the RPC operation are defined within the optional `input` statement. When you execute the RPC, Junos OS invokes the RPC's action script and passes all of the input parameters to the script.

```
input {
   leaf input-param1 {
      type type;
      description description;
   }
   leaf input-param2 {
      type type;
      description description;
   }
   // additional leaf definitions
}
```

NOTE: Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type `empty` when executing the RPC's command in the Junos OS CLI. In earlier releases, input parameters of type `empty` are only supported when executing the RPC in a NETCONF or Junos XML protocol session.

The optional `output` statement encloses the output parameters to the RPC operation. The `output` statement can include one top-level root container. It is a good practice to correlate the name of the root container and the RPC name. For example, if the RPC name is `get-xyz-information`, the container name might be `xyz-information`. Substatements to the `output` statement define nodes under the RPC's `output` node. In the XML output, this would translate into XML elements under the `<rpc-reply>` element.

```
output {
   container output-container-name {
      ...
   }
}
```

Within the root container, you can include `leaf` and `container` statements. Leaf statements describe the data included in the RPC output for that container.

```
output {
   container output-container-name {
      container container-name {
```

```
                leaf output-param1 {
                    type type;
                    description description;
                }
                // additional leaf definitions
            }
        }
    }
```

By default, the format for RPC output is XML. You can also define formatted ASCII output that is displayed when you execute the operational command for that RPC in the CLI or when you request the RPC output in text format.

You define the CLI formatting by defining a `junos-odl:format` statement, which is a Junos OS ODL extension statement.

```
    output {
        container output-container-name {
            container container-name {
                leaf output-param1 {
                    type type;
                    description description;
                    // optional formatting statements
                }
                // additional leaf definitions
                junos-odl:format container-name-format {
                    // CLI formatting for the parent container
                }
            }
            // Additional containers
        }
    }
```

Within the container that defines the CLI formatting, you can customize the RPC’s CLI output by using statements defined in the Junos OS ODL extensions module. For more information about rendering formatted ASCII output, see "Customize YANG RPC Output on Devices Running Junos OS" on page 594. You can also stipulate when the data in a particular container is emitted in an RPC's CLI output. For information about constructing different levels of output for the same RPC, see "Define Different Levels of Output in Custom YANG RPCs for Junos Devices" on page 614.

To use the RPC on a device running Junos OS:

- Download the module and action script to the device.

- Add the files to a new or existing YANG package by issuing the `request system yang add` or `request system yang update` operational command.
- Execute the RPC.
  - To execute the RPC in the CLI, issue the command defined by the `junos:command` statement.
  - To execute the RPC remotely, use the RPC name in an RPC request operation.

You execute the RPC in the CLI by issuing the command defined by the `junos:command` statement. The device displays the RPC output in the CLI format defined by the RPC. If the RPC does not define CLI formatting, by default, no output is displayed for that RPC in the CLI. However, you can still display the XML output for that RPC in the CLI by appending the `| display xml` filter to the command.

For more information about YANG RPCs, see RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, and related RFCs.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
19.2R1	Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type `empty` when executing the RPC's command in the Junos OS CLI.

RELATED DOCUMENTATION

Create Action Scripts for YANG RPCs on Junos Devices | **556**

Use Custom YANG RPCs on Devices Running Junos OS | **569**

Example: Use a Custom YANG RPC to Retrieve Operational Information from Junos Devices | **571**

Understanding Junos OS YANG Extensions for Formatting RPC Output | **590**

Create Action Scripts for YANG RPCs on Junos Devices

SUMMARY

Create a Python or SLAX action script that implements the logic for your custom YANG RPCs.

IN THIS SECTION

You can add YANG data models that define custom remote procedure calls (RPCs) on supported Junos devices. When you add a nonnative YANG RPC to a device, you must also supply an action script that serves as the RPC's handler. The RPC definition references the action script, which is invoked when you execute the RPC. The action script performs the operations and retrieves the information required by the RPC and returns any necessary XML output elements as defined in the RPC `output` statement.

You can write action scripts in Stylesheet Language Alternative SyntaX (SLAX) or Python. SLAX action scripts are similar to SLAX op scripts. They can perform any function available through the RPCs supported by the Junos XML management protocol and the Junos XML API. Python action scripts can leverage all of the features and constructs in the Python language, which provides increased flexibility over SLAX scripts. In addition, Python action scripts support Junos PyEZ APIs, which facilitate executing RPCs and performing operational and configuration tasks on Junos devices. Python scripts can also leverage the `lxml` library, which simplifies XPath handling.

This topic discusses how to create an action script, including how to parse the RPC input arguments, access operational and configuration data in the script, emit the XML output, and validate and load the script on a device.

Action Script Boilerplate

IN THIS SECTION

SLAX Script Boilerplate

SLAX action scripts must include the necessary boilerplate for both basic script functionality as well as any optional functionality used within the script such as the Junos OS *extension functions* and *named templates*. In addition, the script must declare all RPC input parameters using the `param` statement. The SLAX action script boilerplate is as follows:

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "/var/db/scripts/import/junos.xsl";

param $input-param1;
param $input-param2;

match / {
    <action-script-results> {
        /* insert your code here */
    }
}
```

Python Script Boilerplate

Python action scripts must include an interpreter directive line that specifies the Python version used to execute the script. Table 31 on page 557 outlines the interpreter directive lines you can use in the different releases.

Table 31: Python Action Script Interpreter Directive Lines

Python Version	Interpreter Directive Lines	Supported Releases

Python 3	`#!/usr/bin/python3`
	or
	`#!/usr/bin/env python3`
		Junos OS Release 20.2R1 and later
		
		Junos OS Evolved Release 21.1R1 and later

Table 31: Python Action Script Interpreter Directive Lines *(Continued)*

Python Version	Interpreter Directive Lines	Supported Releases

Python 2.7	`#!/usr/bin/python`
	or
	`#!/usr/bin/env python`
		Junos OS Release 20.1 and earlier
		
		Junos OS Evolved Release 22.2 and earlier

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

In addition, Python action scripts should import any libraries, modules, or objects that are used in the script. For example, in addition to standard Python libraries, Python action scripts might import the following:

- `jcs` library—Enables the script to use Junos OS *extension functions* and Junos OS *named template* functionality in the script.
- `jnpr.junos` module and classes—Enables the script to use Junos PyEZ.
- `lxml` library—Simplifies XPath handling.

For example:

```
#!/usr/bin/python3
import jcs
from jnpr.junos import Device
from lxml import etree
```

Parsing RPC Input Arguments

IN THIS SECTION

Input Argument Overview

An RPC can define input parameters using the optional `input` statement. When you execute an RPC and provide input arguments, Junos OS invokes the RPC's action script and passes those arguments to the script. In a Python or SLAX action script, you can access the RPC input arguments in the same manner as you would access command-line arguments for a normal Python script or a Junos OS SLAX op script, respectively.

Consider the following `input` statement for the `get-host-status` RPC:

```
rpc get-host-status {
   description "RPC example to retrieve host status";

   junos:command "show host-status" {
      junos:action-execute {
         junos:script "rpc-host-status.py";
      }
   }

   input {
     leaf hostip {
       description "IP address of the target host";
       type string;
     }
     leaf level {
       type enumeration {
         enum brief {
           description "Display brief output";
         }
         enum detail {
           description "Display detailed output";
         }
       }
     }
     leaf test {
       description "empty argument";
       type empty;
     }
   }
   ...
```

You can execute an RPC in the CLI or through a NETCONF or Junos XML protocol session. For example, you might execute the following command in the CLI:

```
user@host> show host-status hostip 198.51.100.1 level detail test
```

Similarly, you might execute the following RPC in a remote session:

```
<rpc>
    <get-host-status>
        <hostip>198.51.100.1</hostip>
        <level>detail</level>
        <test/>
    </get-host-status>
</rpc>
```

When you execute the command or RPC, the device invokes the action script and passes in the arguments. The following sections discuss how to process the arguments in the SLAX or Python action script.

NOTE: Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type `empty` when executing the RPC's command in the Junos OS CLI, and the value passed to the action script is the parameter name. In earlier releases, input parameters of type `empty` are only supported when executing the RPC in a NETCONF or Junos XML protocol session, and the value passed to the action script is the string `'none'`.

SLAX Script Input Arguments

In SLAX action scripts, you must declare input parameters using the `param` statement. The parameter names must be identical to the parameter names defined in the YANG module.

When invoked, the script assigns the value for each argument to the corresponding parameter, which you can then reference throughout the script. You must include the dollar sign ($) symbol both when you declare the parameter and when you access its value. If a parameter is type `empty`, the parameter name is passed in as its value.

```
param $hostip;
param $level;
param $test;
```

NOTE: For more information about SLAX parameters, see *SLAX Parameters Overview* in the Automation Scripting User Guide.

Python Script Input Arguments

For Python action scripts, the arguments are passed to the script as follows:

- The first argument is always the action script's file path.
- The next arguments in the list are the name and value for each input parameter supplied by the user.

  The argument name is passed in as follows:

  - In Junos OS Release 21.1 and earlier, the device passes in the name of the argument.
  - In Junos OS Release 21.2R1 and later, the device prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

  NOTE: When you execute the RPC's command in the CLI, the arguments are passed to the script in the order given on the command line. In a NETCONF or Junos XML protocol session, the order of arguments in the XML is arbitrary, so the arguments are passed to the script in the order that they are declared in the RPC `input` statement.

- The last two arguments in the list, which are supplied by the system and not the user, are `'rpc_name'` and the name of the RPC.

The following sections discuss how to handle the arguments that are passed to Python action scripts in the different releases.

Python Action Scripts (21.2R1 or later)

Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes the input argument names to the Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names. This syntax enables you to use standard command-line parsing libraries to handle the arguments.

For the previous YANG RPC example, the action script's `sys.argv` input argument list is:

```
['/var/db/scripts/action/rpc-host-status.py', '--hostip', '198.51.100.1', '--level', 'detail',
'--test', 'test', '--rpc_name', 'get-host-status']
```

The following sample Python code uses the argparse library to handle the arguments. In this case, the parser must also account for the rpc_name argument that the system passes to the script.

```
#!/usr/bin/python3
import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--hostip', required=True)
parser.add_argument('--level', required=False, default='brief')
parser.add_argument('--test', required=False)
parser.add_argument('--rpc_name', required=True)
args = parser.parse_args()

# access argument values by using args.hostip, args.level, and args.test
```

Python Action Scripts (21.1 and earlier)

In Junos OS Release 21.1 and earlier, the device passes the input argument names to the Python action script exactly as they are given in the command or RPC. You can access the input arguments through the sys.argv list.

For the previous YANG RPC example, the action script's sys.argv input argument list is:

```
['/var/db/scripts/action/rpc-host-status.py', 'hostip', '198.51.100.1', 'level', 'detail',
'test', 'test', 'rpc_name', 'get-host-status']
```

The following sample Python code demonstrates one way to extract the value for each argument from the sys.arv list for the example RPC. The example first defines a dictionary containing the possible argument names as keys and a default value for each argument. The code then checks for each key in the sys.argv list and retrieves the index of the argument name in the list, if it is present. The code then extracts the argument's value at the adjacent index position, and stores it in the dictionary for the appropriate key. This method ensures that if the arguments are passed to the script in a different order during execution, the correct value is retrieved for a given argument.

```
import sys

# Define default values for arguments
args = {'hostip': None, 'level': 'brief', 'test': None}

# Retrieve user input and store the values in the args dictionary
```

```
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]
```

Retrieving Operational and Configuration Data

Action scripts can retrieve operational and configuration data from a device running Junos OS and then parse the data for necessary information. SLAX action scripts can retrieve information from the device by executing RPCs supported by the Junos XML protocol and the Junos XML API. Python action scripts can retrieve operational and configuration information by using Junos PyEZ APIs or by using the `cli -c '`*`command`*`'` to execute CLI commands in the action script as you would from the shell. To retrieve operational information with the `cli -c` method, include the desired operational command. To retrieve configuration information, use the `show configuration` command.

The following SLAX snippet executes the `show interfaces` command on the local device by using the equivalent `<get-interface-information>` request tag:

```
var $rpc = <get-interface-information>;
var $out = jcs:invoke($rpc);
/* parse for relevant information and return as XML tree for RPC output */
```

The following Python code uses Junos PyEZ to execute the `get_interface_information` RPC, which is equivalent to the `show interfaces` CLI command:

```
#!/usr/bin/python3
from jnpr.junos import Device
from lxml import etree

with Device() as dev:
    res = dev.rpc.get_interface_information()
    # parse for relevant information and return as XML tree for RPC output
```

NOTE: For information about using Junos PyEZ to execute RPCs on devices running Junos OS, see Use Junos PyEZ to Execute RPCs on Junos Devices.

The following Python code executes the `show interfaces | display xml` command and converts the string output into an XML tree that can be parsed for the required data using XPath constructs:

```
#!/usr/bin/python3
import subprocess
from lxml import etree

cmd = ['cli', '-c', 'show interfaces | display xml']
proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
tmp = proc.stdout.read()
root = etree.fromstring(tmp.strip())
# parse for relevant information and return as XML tree for RPC output
```

Emitting the RPC XML Output

An RPC can define output elements using the optional `output` statement. The action script must define and emit any necessary XML elements for the RPC output. The XML hierarchy emitted by the script should reflect the tree defined by the containers and leaf statements in the definition of the RPC `output` statement. To return the XML output, the action script must emit the RPC output hierarchy, and only the output hierarchy. SLAX scripts must use the `copy-of` statement to emit the XML, and Python scripts can use `print` statements.

For example, consider the following YANG RPC `output` statement:

```
     output {
       container host-status-information {
         container host-status-info {
           leaf host {
             type string;
             description "Host IP";
           }
           leaf status {
             type string;
             description "Host status";
           }
           leaf date {
             type string;
             description "Date and time";
           }
         }
```

```
        }
      }
```

The action script must generate and emit the corresponding XML output, for example:

```
<host-status-information>
  <host-status-info>
    <host>198.51.100.1</host>
    <status>Active</status>
    <date>2016-10-10</date>
  </host-status-info>
  <host-status-info>
    <host>198.51.100.2</host>
    <status>Inactive</status>
    <date>2016-10-10</date>
  </host-status-info>
</host-status-information>
```

After retrieving the values for the required output elements, a Python script might emit the XML output hierarchy by using the following code:

```
from lxml import etree
...

xml = '''
<host-status-information>
  <host-status-info>
    <host>{0}</host>
    <status>{1}</status>
    <date>{2}</date>
  </host-status-info>
</host-status-information>
'''.format(hostip, pingstatus, now)

tree = etree.fromstring(xml)
print (etree.tostring(tree))
```

Similarly, a SLAX action script might use the following:

```
var $node = {
    <host-status-information> {
        <host-status-info> {
            <host> $ip;
            <status> $pingstatus;
            <date> $date;
        }
    }
}
copy-of $node;
```

Validating and Loading Action Scripts on a Device

In your YANG RPC definition, you specify the RPC's action script by including the `junos:command` and `junos:action-execute` statements and the `junos:script` substatement. The `junos:script` statement takes the action script's filename as its value. You must define one and only one action script for each RPC. For example:

```
rpc rpc-name {
   ...
   junos:command "show sw-info" {
      junos:action-execute {
         junos:script "sw-info.py";
      }
   }
   ...
}
```

NOTE: YANG modules that define RPCs for Junos devices must import the Junos OS DDL extensions module.

Python action scripts must meet the following requirements before you can execute the scripts on the device:

- File owner is either root or a user in the Junos OS `super-user` login class.
- Only the file owner has write permission for the file.
- Script includes an interpreter directive line as outlined in "Action Script Boilerplate" on page 556.
- The device configuration includes the `language python` or `language python3` statement at the `[edit system scripts]` hierarchy level to enable the execution of unsigned Python scripts.

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

NOTE: Users can only execute unsigned Python scripts on devices running Junos OS when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

You can validate the syntax of an action script in the CLI by issuing the `request system yang validate action-script` command and providing the path to the script. For example:

```
user@host> request system yang validate action-script /var/tmp/sw-info.py
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

To use an action script, you must load it onto the device with the YANG module that contains the corresponding RPC. You use the `request system yang add` or `request system yang update` commands to add YANG modules and their action scripts to a new or existing YANG package on the device. After you add the modules and action scripts to the device, you can execute your custom RPCs. When you execute an RPC, the device invokes the referenced script.

Troubleshooting Action Scripts

By default, action scripts log informational trace messages when the script executes. You can view the trace messages to verify that the RPC invoked the script and that the script executed correctly. If the script fails for any reason, the trace file logs the errors.

Junos OS

To view action script trace messages on devices running Junos OS, view the contents of the **action.log** trace file.

```
user@host> show log action.log
```

Junos OS Evolved

To view action script trace messages on devices running Junos OS Evolved, view the cscript application trace messages, which include trace data for all script types.

```
user@host> show trace application cscript
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
22.3R1-EVO	Starting in Junos OS Evolved Release 22.3R1, Junos OS Evolved uses Python 3 to execute YANG action and translation scripts.
21.2R1 and 21.2R1-EVO	Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.
20.2R1	Starting in Junos OS Release 20.2R1, Junos OS uses Python 3 to execute YANG action and translation scripts.
19.3R1	Starting in Junos OS Release 19.3R1, devices running Junos OS with Upgraded FreeBSD support using IPv6 in Python action scripts.
19.2R1	Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type `empty` when executing the RPC's command in the Junos OS CLI, and the value passed to the action script is the parameter name.

RELATED DOCUMENTATION

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Use Custom YANG RPCs on Devices Running Junos OS | **569**

Example: Use a Custom YANG RPC to Retrieve Operational Information from Junos Devices | **571**

Use Custom YANG RPCs on Devices Running Junos OS

You can add YANG data models that define custom RPCs on supported devices running Junos OS or Junos OS Evolved. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices.

To add an RPC to a device, download the YANG module that defines the RPC, along with any required action scripts to the device. Add the files to a new or existing YANG package by issuing the `request system yang add` or `request system yang update` operational command. For detailed information about adding YANG modules to Junos devices, see "Manage YANG Packages, Modules, and Scripts on Junos Devices" on page 527.

NOTE: When you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules.

After you add the modules and action scripts to the device, you can execute the RPC either locally, provided that the RPC definition includes the `junos:command` statement, or remotely. To execute an RPC in the Junos OS CLI, issue the command defined by the RPC's `junos:command` statement. To execute an RPC remotely, use the RPC name in an RPC request operation.

Consider the following YANG module and RPC definition:

```
module sw-info {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix rpc-cli;

  import junos-extension {
    prefix junos;
  }

  rpc get-sw-info {
    description "Show software information";
    junos:command "show sw-info" {
       junos:action-execute {
         junos:script "sw-info.py";
      }
    }
    input {
      leaf routing-engine {
        type string;
        description "Routing engine for which to display information";
      }
      ...
    }
    output {
      ...
    }
  }
}
```

To execute this RPC in the Junos OS CLI, issue the `show sw-info` command defined by the `junos:command` statement, and include any required or optional input parameters. For example:

```
user@host> show sw-info routing-engine re0
```

To execute this RPC remotely, send an RPC request that uses the RPC name for the request tag, and include any required or optional input parameters.

```
<rpc>
   <get-sw-info>
     <routing-engine>re0</routing-engine>
   </get-sw-info>
</rpc>
```

When you execute a custom RPC, the device invokes the action script that you defined in the `junos:script` statement, which in this example is the **sw-info.py** script. An RPC's action script should emit any necessary XML elements for that RPC's output.

You execute an RPC in the Junos OS CLI by issuing the command defined by the `junos:command` statement. By default, the CLI displays the RPC output, if there is any, using the CLI formatting defined by the RPC. If the RPC does not define CLI formatting, the device does not display any output for that RPC in the CLI. However, you can still display the RPC's XML output in the CLI by appending `| display xml` to the command.

```
user@host> show sw-info routing-engine re0 | display xml
```

When you execute an RPC remotely, the RPC output defaults to XML. However, you can specify a different output format by including the `format` attribute in the opening request tag of the RPC. To display

CLI formatting, provided that the RPC defines this format, set the `format` attribute to `text` or `ascii`. To display the output in JavaScript Object Notation (JSON), set the `format` attribute to `json`. For example:

```
<rpc>
   <get-sw-info format="text">
     <routing-engine>re0</routing-engine>
   </get-sw-info>
</rpc>
```

RELATED DOCUMENTATION

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Create Action Scripts for YANG RPCs on Junos Devices | **556**

Example: Use a Custom YANG RPC to Retrieve Operational Information from Junos Devices | **571**

Example: Use a Custom YANG RPC to Retrieve Operational Information from Junos Devices

IN THIS SECTION

You can add YANG data models that define custom RPCs on Junos devices. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices. This example presents a custom RPC and action script that retrieve operational information from the device and display customized CLI output.

The example adds the RPC to the Junos OS schema on the device. When you execute the RPC in the CLI, it prints the name and operational status for the requested physical interfaces.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS or device running Junos OS Evolved
- Device supports loading custom YANG data models

Overview of the RPC and Action Script

The YANG module in this example defines a custom RPC to return the name and operational status of certain physical interfaces. The YANG module `rpc-interface-status` is saved in the **rpc-interface-status.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the CLI output.

The module defines the `get-interface-status` RPC. The `<get-interface-status>` request tag is used to remotely execute the RPC on the device. In the RPC definition, the `junos:command` statement defines the command that is used to execute the RPC in the CLI, which in this case is `show intf status`.

The `junos:action-execute` and `junos:script` statements define the action script that is invoked when you execute the RPC. This example uses a Python action script named **rpc-interface-status.py** to retrieve the information required by the RPC and return the XML output elements as defined in the RPC `output` statement.

```
rpc get-interface-status {
   description "RPC example to retrieve interface status";

   junos:command "show intf status" {
      junos:action-execute {
        junos:script "rpc-interface-status.py";
      }
   }
  ...
```

The RPC has one input parameter named `match`, which determines the interfaces to include in the output. When you execute the RPC, you include a string that matches on the required interfaces, for example ge-0*. An empty string ("") matches on all interfaces. The action script defines the default value for `match` as an empty string, so if the user omits this argument, the output includes information for all interfaces.

```
input {
  leaf match {
    description "Requested interface match condition";
    type string;
  }
}
```

The RPC also defines the output nodes that the corresponding action script must emit. The root node is the `<interface-status-info>` element, which contains zero or more `<status-info>` elements that enclose the `<interface>` and `<status>` nodes for a matched interface. The `junos-odl:format interface-status-info-format` statement defines the formatting for the CLI text output. The output XML tree does not include this node.

```
output {
  container interface-status-info {
    list status-info {
      leaf interface {
        type string;
        description "Physical interface name";
      }
      leaf status {
        type string;
        description "Operational status";
      }
      junos-odl:format interface-status-info-format {
        ...
      }
    }
  }
}
```

This example presents two versions of the Python action script. The scripts demonstrate different means to retrieve the operational command output, but both scripts emit identical RPC output. The first action script uses the Python `subprocess` module to execute the `show interfaces` *`match-value`* `| display xml` command and then converts the string output into XML. The second action script uses Junos PyEZ to execute the RPC equivalent of the `show interfaces` *`match-value`* command. Both scripts use identical code to parse the command output and extract the name and operational status for each physical interface. The scripts construct the XML for the RPC output and then print the output, which returns the information back to the device. The XML tree must exactly match the hierarchy defined in the RPC.

NOTE: Junos devices define release-dependent namespaces for many of the elements in the operational output, including the `<interface-information>` element. In order to make the RPC release independent, the code uses the `local-name()` function in the XPath expressions for these elements. You might choose to include the namespace mapping as an argument to `xpath()` and qualify the elements with the appropriate namespace.

You add the module containing the RPC and the action script file to the device as part of a new YANG package named `intf-rpc`.

YANG Module

IN THIS SECTION

YANG Module

The YANG module, **rpc-interface-status.yang**, defines: the RPC, the command that you use to execute the RPC in the CLI, and the name of the action script to invoke when you execute the RPC. The base name of the file must match the module name.

```
/*
* Copyright (c) 2024 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-interface-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix rpc-cli;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
```

```
}

organization
  "Juniper Networks, Inc.";

description
  "Junos OS YANG module for RPC example";

rpc get-interface-status {
   description "RPC example to retrieve interface status";

   junos:command "show intf status" {
      junos:action-execute {
        junos:script "rpc-interface-status.py";
      }
   }

   input {
     leaf match {
       description "Requested interface match condition";
       type string;
     }
   }
   output {
     container interface-status-info {
       list status-info {
         leaf interface {
           type string;
           description "Physical interface name";
         }
         leaf status {
           type string;
           description "Operational status";
         }
         junos-odl:format interface-status-info-format {
           junos-odl:header "Physical Interface - Status\n";
           junos-odl:indent 5;
           junos-odl:comma;
           junos-odl:space;
           junos-odl:line {
             junos-odl:field "interface";
             junos-odl:field "status";
           }
```

```
            }
          }
        }
      }
   }
 }
```

Action Script

IN THIS SECTION

The corresponding action script is **rpc-interface-status.py**. This example presents two action scripts that use different means to retrieve the data. One script uses the Python `subprocess` module. The other script uses the Junos PyEZ library. Both scripts emit the same RPC XML output.

NOTE: Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

Action Script (Using `subprocess`)

The following action script uses the Python `subprocess` module to execute the operational command and retrieve the data. This example provides two versions of the script, which appropriately handle the script's command-line arguments for the different releases.

Junos OS Release 21.1 and earlier

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
import subprocess
from lxml import etree
```

```
def get_device_info(cmd):
    """
    Execute Junos OS operational command and parse output
    :param: str cmd: operational command to execute
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
    tmp = proc.stdout.read()
    root = etree.fromstring(tmp.strip())

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(cmd):
    """
    Generate the XML tree for the RPC output
    :param: str cmd: operational command from which to retrieve data
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')
    intf_list_xml = get_device_info(cmd)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml


def main():

    args = {'match': ""}
    for arg in args.keys():
        if arg in sys.argv:
            index = sys.argv.index(arg)
            args[arg] = sys.argv[index+1]

    # define the operational command from which to retrieve information
    cli_command = 'show interfaces ' + args['match'] + ' | display xml'
    cmd = ['cli', '-c', cli_command]

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(cmd)

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True, encoding='unicode'))


if __name__ == '__main__':

    main()
```

Junos OS Release 21.2R1 and later

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import subprocess
import argparse
from lxml import etree

def get_device_info(cmd):
    """
    Execute Junos OS operational command and parse output
    :param: str cmd: operational command to execute
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
    tmp = proc.stdout.read()
    root = etree.fromstring(tmp.strip())

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(cmd):
    """
    Generate the XML tree for the RPC output
    :param: str cmd: operational command from which to retrieve data
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')

    intf_list_xml = get_device_info(cmd)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml


def main():

    parser = argparse.ArgumentParser(description='This is a demo script.')
    parser.add_argument('--match', required=False, default='')
    parser.add_argument('--rpc_name', required=True)
    args = parser.parse_args()

    # define the operational command from which to retrieve information
    cli_command = 'show interfaces ' + args.match + ' | display xml'
    cmd = ['cli', '-c', cli_command]

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(cmd)

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True, encoding='unicode'))


if __name__ == '__main__':

    main()
```

Action Script (Using Junos PyEZ)

The following action script uses Junos PyEZ to execute the operational command and retrieve the data. This example provides two versions of the script, which appropriately handle the script's command-line arguments for the different releases.

Junos OS Release 21.1 and earlier

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
from jnpr.junos import Device
from jnpr.junos.exception import *
from lxml import etree
```

```
def get_device_info(match):
    """
    Execute Junos OS operational command and parse output
    :param: str match: interface match condition
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    try:
        with Device() as dev:
            if (match == ""):
                root = dev.rpc.get_interface_information( )
            else:
                root = dev.rpc.get_interface_information(interface_name=match)
    except Exception:
        sys.exit()

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(match):
    """
    Generate the XML tree for the RPC output
    :param: str match: interface match condition
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')

    intf_list_xml = get_device_info(match)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml


def main():

    args = {'match': ""}

    for arg in args.keys():
        if arg in sys.argv:
            index = sys.argv.index(arg)
            args[arg] = sys.argv[index+1]

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(args['match'])

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True, encoding='unicode'))

if __name__ == '__main__':

    main()
```

Junos OS Release 21.2R1 and later

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import sys
import argparse
from jnpr.junos import Device
from jnpr.junos.exception import *
from lxml import etree
```

```
def get_device_info(match):
    """
    Execute Junos OS operational command and parse output
    :param: str match: interface match condition
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    try:
        with Device() as dev:
            if (match == ""):
                root = dev.rpc.get_interface_information( )
            else:
                root = dev.rpc.get_interface_information(interface_name=match)
    except Exception:
        sys.exit()

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(match):
    """
    Generate the XML tree for the RPC output
    :param: str match: interface match condition
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')

    intf_list_xml = get_device_info(match)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml



def main():

    parser = argparse.ArgumentParser(description='This is a demo script.')
    parser.add_argument('--match', required=False, default='')
    parser.add_argument('--rpc_name', required=True)
    args = parser.parse_args()

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(args.match)

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True, encoding='unicode'))

if __name__ == '__main__':

    main()
```

Enable the Execution of Python Scripts

To enable the device to execute unsigned Python scripts:

1. Configure the `language python` or `language python3` statement, as appropriate for the Junos OS release.

   ```
   [edit]
   user@host# set system scripts language (python | python3)
   ```

   NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

2. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

Load the RPC on the Device

To add the RPC and action script to the Junos schema:

1. Download the YANG module and action script to the Junos device.
2. Ensure that the Python action script meets the following requirements:
   - File owner is either root or a user in the Junos OS `super-user` login class.
   - Only the file owner has write permission for the file.
   - Script includes the appropriate interpreter directive line as outlined in "Create Action Scripts for YANG RPCs on Junos Devices" on page 556.
3. (Optional) Validate the syntax for the YANG module and action script.

```
user@host> request system yang validate module /var/tmp/rpc-interface-status.yang action-
script /var/tmp/rpc-interface-status.py
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

4. Add the YANG module and action script to a new YANG package.

```
user@host> request system yang add package intf-rpc module /var/tmp/rpc-interface-status.yang
action-script /var/tmp/rpc-interface-status.py
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
```

```
Restarting mgd ...
```

5. The system prompts you to restart the Junos OS CLI. Press `Enter` to accept the default value of `yes`, or type **yes** and press `Enter`.

```
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

Verify the RPC

IN THIS SECTION

Purpose

Verify that the RPC works as expected.

Action

From operational mode, execute the RPC in the CLI by issuing the command defined by the `junos:command` statement in the RPC definition, and include the `match` input argument. In this example, the match argument is used to match on all interfaces that start with ge-0.

```
user@host> show intf status match ge-0*
Physical Interface - Status
     ge-0/0/0, up
     ge-0/0/1, up
     ge-0/0/2, up
     ge-0/0/3, up
     ge-0/0/4, up
     ge-0/0/5, up
     ge-0/0/6, up
     ge-0/0/7, up
     ge-0/0/8, up
     ge-0/0/9, up
     ge-0/1/0, up
     ge-0/1/1, up
     ge-0/1/2, up
     ge-0/1/3, up
     ge-0/1/4, up
     ge-0/1/5, up
     ge-0/1/6, up
     ge-0/1/7, up
     ge-0/1/8, up
     ge-0/1/9, up
```

You can also adjust the match condition to return different sets of interfaces. For example:

```
user@host> show intf status match *e-0/*/0
Physical Interface - Status
     ge-0/0/0, up
     pfe-0/0/0, up
     ge-0/1/0, up
     xe-0/2/0, up
     xe-0/3/0, up
```

To return the same output in XML format, append the `| display xml` filter to the command.

```
user@host> show intf status match *e-0/*/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/24.4R1/junos">
    <interface-status-info>
          <status-info>
            <interface>ge-0/0/0</interface>
            <status>up</status>
          </status-info>
          <status-info>
            <interface>pfe-0/0/0</interface>
            <status>up</status>
          </status-info>
          <status-info>
            <interface>ge-0/1/0</interface>
```

```
              <status>up</status>
            </status-info>
            <status-info>
              <interface>xe-0/2/0</interface>
              <status>up</status>
            </status-info>
            <status-info>
              <interface>xe-0/3/0</interface>
              <status>up</status>
            </status-info>
      </interface-status-info>
      <cli>
          <banner></banner>
      </cli>
  </rpc-reply>
```

NOTE: To match on all interfaces, either omit the `match` argument or set the value of the argument to an empty string ("").

Meaning

When you execute the RPC, the device invokes the action script. The action script executes the operational command to retrieve the interface information from the device and parses the output for the required information. The script then prints the XML hierarchy for the RPC output as defined in the RPC `output` statement. When you execute the RPC in the CLI, the device uses the CLI formatting defined in the RPC to convert the XML output into the displayed CLI output. To return the original XML output, append the `| display xml` filter to the command.

NOTE: When you execute the RPC remotely using the RPC request tag, the default format for the output is XML.

Troubleshoot RPC Execution Errors

IN THIS SECTION

Problem

Description

When you execute the RPC, the device generates the following error:

```
error: open failed: /var/db/scripts/action/rpc-interface-status.py: Permission denied
```

Cause

The user who invoked the RPC does not have the necessary permissions to execute the corresponding Python action script.

Solution

Users can only execute unsigned Python scripts on Junos devices when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

Verify whether the script has the necessary permissions for that user to execute the script, and adjust the permissions, if appropriate. If you update the permissions, you must also update the YANG package in order for this change to take effect. For example:

```
admin@host> file list ~ detail
-rw-------  1 admin    wheel  2215 Apr 20 11:36 rpc-interface-status.py
```

```
admin@host> file change-permission rpc-interface-status.py permission 644
admin@host> file list ~ detail
-rw-r--r--  1 admin    wheel  2215 Apr 20 11:36 rpc-interface-status.py
```

```
admin@host> request system yang update intf-rpc action-script /var/tmp/rpc-interface-status.py
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1 and 21.2R1-EVO	Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

RELATED DOCUMENTATION

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Create Action Scripts for YANG RPCs on Junos Devices | **556**

Use Custom YANG RPCs on Devices Running Junos OS | **569**

Manage YANG Packages, Modules, and Scripts on Junos Devices | **527**

Understanding Junos OS YANG Extensions for Formatting RPC Output

Junos OS natively supports XML for the operation and configuration of devices running Junos OS and devices running Junos OS Evolved. The Junos OS infrastructure and CLI communicate using XML. When you issue an operational command in the CLI, the CLI converts the command into XML for processing. After processing, Junos OS returns the output in the form of an XML document, which the CLI converts back into text format for display. Remote client applications also use XML-based data encoding for operational and configuration requests on devices running Junos OS.

The Junos OS Output Definition Language (ODL) defines the transformation of XML-tagged data into the formatted ASCII output that the device displays when you execute a command in the CLI or request RPC output in text format. The Junos OS ODL extensions module defines YANG extensions for the ODL, which you can include in custom YANG RPCs to translate the XML RPC reply into formatted ASCII output.

The YANG RPC `output` statement defines output parameters to the RPC operation. Within the RPC `output` statement, you can include ODL extension statements to customize the RPC's output. Table 32 on page 591 outlines the available statements, provides a brief description of each statement's formatting impact, and specifies the locations where you can define the statement within the RPC `output` statement.

You include some ODL extension statements under the leaf statement that defines the data. You include other ODL extension statements within the output container or at various levels within the `format` statement, which defines the CLI formatting. The placement of a statement within the `format` statement determines the statement's scope. The scope might apply to a single field, all fields in a line, or all fields in all lines of output. Statements that you can define at any level in the `format` statement can be included in the following locations:

- At the top level as a direct child of the `format` statement
- Directly under the `line` statement
- Within a `field` statement

Table 32: Statements in the Junos OS ODL Extensions Module

Statement	Description	Placement Within RPC `output` Statement
`blank-line`	Insert a blank line between each repetition of data when the RPC reply returns the same set of information for multiple entities.	`format` statement (top level)
`capitalize`	Capitalize the first word of a node's value in an output field.	`format` statement (any level)
`colon`	Insert a colon following the node's label in an output field. This statement is only used in conjunction with the `leading` statement to insert the formal name of the node, as defined by the `formal-name` statement, and a colon before the value of the node in the output field.	`format` statement (any level)
`comma`	Insert a comma after a node's value in an output field.	`format` statement (any level)
`default-text`	Specify the text to display when the node corresponding to an output field is missing.	`field` statement
`explicit`	Direct the renderer to display a value that is unrelated to the node name or its contents. This statement is used in Junos OS RPCs only and cannot be included in custom RPCs.	–
`field`	Map a leaf node in the output tree to a field in the formatted ASCII output.	`line` statement

Table 32: Statements in the Junos OS ODL Extensions Module *(Continued)*

Statement	Description	Placement Within RPC output Statement
fieldwrap	Wrap a field’s complete contents to the following line when the current line is wider than the screen. Omitting this statement causes the output to wrap without regard for appropriate word breaks or the prevailing margin.	field statement
float	Enable the value in a field to move to the left into an empty field. Use this statement to indicate subsequent mutually exclusive values for a set of adjacent fields so that only the leftmost field includes one of these possible values. If the leftmost field is not populated by the first value, a value mapped to a subsequent field that includes the float statement can move into the empty field.	field statement
formal-name	Define the label that precedes a node’s value in an output field whenever the field for that node includes the leading statement in the formatting instructions.	leaf node
format	Define the CLI formatting for the parent container within the RPC output statement.	output container or as a substatement to the style statement.
header	Define a header row in the CLI output.	format statement (top level)
header-group	Require that only the first header string as defined by the header statement be emitted in the CLI output for that header group.	format statement (top level)
indent	Indent all lines other than the header row by the specified number of spaces in the CLI output.	format statement (top level)
leading	Insert a label, which is defined by the formal-name statement in the definition of a leaf node, before the node’s value in an output field.	format statement (any level)

Table 32: Statements in the Junos OS ODL Extensions Module *(Continued)*

Statement	Description	Placement Within RPC `output` Statement
`line`	Define the group of fields that comprises a single line of output.	`format` statement (top level)
`no-line-break`	Display multiple values on the same line in the case where multiple entities with the same tag names are emitted.	`format` statement (top level)
`picture`	Graphically specify the placement, justification, and width of the columns in a table in the RPC's formatted ASCII output.	`format` statement (top level)
`space`	Insert a space after the node's value in an output field. If the `space` statement is used in conjunction with the `comma` statement, the output inserts a comma and then a space after the node's value, in that order.	`format` statement (any level)
`style`	Define a format, or style, for the RPC output. Use this statement in conjunction with an enumerated input parameter that defines the names for each style. Define this statement with the appropriate style name to specify the CLI formatting for that style.	output container
`template`	Explicitly define the format for an output field, including the output string and the placement of the node's value within that string. Use `%s` or `%d` to indicate the placement of the node's string or integer value, respectively, within the output string. If a leaf statement defines both a `template` and a `formal-name` statement, and the corresponding field's formatting instructions include the `leading` statement, the output displays the text defined for the `formal-name` statement and not the text defined for the `template` statement.	`leaf` node

Table 32: Statements in the Junos OS ODL Extensions Module *(Continued)*

Statement	Description	Placement Within RPC `output` Statement
`truncate`	Truncate a node's value to fit the field width defined by the `picture` statement if the node's contents would otherwise exceed the width of the field.	`field` statement
`wordwrap`	Wrap some of the field to the following line when the current line is wider than the screen. This statement should only be used for fields in the rightmost column of a table.	`field` statement

For more information about the structure of YANG RPCs, see "Create Custom RPCs in YANG for Devices Running Junos OS" on page 549.

RELATED DOCUMENTATION

Customize YANG RPC Output on Devices Running Junos OS | **594**

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Define Different Levels of Output in Custom YANG RPCs for Junos Devices | **614**

Customize YANG RPC Output on Devices Running Junos OS

SUMMARY

Learn about the Junos OS ODL extension statements that you can include in custom YANG RPCs to define the CLI format for RPC output.

IN THIS SECTION

You can create custom RPCs in YANG for devices running Junos OS or devices running Junos OS Evolved. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for specific operational tasks on your devices.

When you execute an RPC on a device running Junos OS, it returns the RPC reply as an XML document. The Junos OS Output Definition Language (ODL) defines the transformation of XML data into formatted ASCII output. The device displays this text output when you execute a command in the CLI or request RPC output in text format. The Junos OS ODL extensions module defines YANG extensions for the Junos OS ODL, which you can include in custom RPCs to specify the CLI formatting for the output. For a summary of all the statements and their placement within the RPC `output` statement, see "Understanding Junos OS YANG Extensions for Formatting RPC Output" on page 590.

The following sections outline how to use the Junos OS ODL extension statements. Closely related statements are presented in the same section, and in some instances, a statement might be included in more than one section. The examples assume that the enclosing YANG module imports the Junos OS ODL extensions module and binds it to the `junos-odl` prefix.

blank-line

The `blank-line` statement inserts a line between each repetition of data when the RPC reply returns the same set of information for multiple entities. For example, if the RPC reply returns data for multiple interfaces, the formatted ASCII output inserts a blank line between each interface's set of data.

```
Physical interface: ge-0/0/0, Enabled, Physical link is Up
  Interface index: 150, SNMP ifIndex: 528
  ...
```

```
Physical interface: ge-0/0/1, Enabled, Physical link is Up
  Interface index: 151, SNMP ifIndex: 529
  ...
```

To insert a blank line between each entity's data set, include the `blank-line` statement directly under the `format` statement.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         // leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:blank-line;
            // CLI formatting
          }
      }
   }
}
```

capitalize

The `capitalize` statement capitalizes the first word of a node's value in an output field. It does not affect the capitalization of a node's formal name. For example, if the RPC output includes a `state` node with the value `online`, the `capitalize` statement capitalizes the value in the output.

```
State: Online
```

To capitalize the first word of the node's value, include the `capitalize` statement within the `format` statement. The placement of the statement determines the statement's scope and whether it affects a single field, all fields in a single line, or all lines.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf state {
            junos-odl:formal-name "State";
            type string;
            description "Interface state";
         }
         junos-odl:format xyz-information-format {
            junos-odl:header "xyz information\n";
            junos-odl:line {
               junos-odl:field "state" {
                  junos-odl:leading;
                  junos-odl:colon;
                  junos-odl:capitalize;
               }
            }
         }
      }
   }
}
```

colon, formal-name, and leading

A node’s formal name, or label, is the text that precedes a node’s contents in the output. To create a label for a node, you must include the `formal-name` statement in the definition of the leaf node. Additionally, you must include the `leading` statement in the formatting instructions for that node’s output field.

In the following example, the `version` node has the formal name `Version`:

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:formal-name "Version";
            type string;
            description "Version";
         }
         ...
      }
   }
}
```

The `colon` statement inserts a colon after the node’s label in an output field. If the formatting instructions include both the `colon` and `leading` statements, the output inserts the node’s label and a colon before the node’s value. For example:

```
Version: value
```

To insert the label and a colon in the output field, include the `leading` and `colon` statements within the `format` statement. The placement of the statements determines the scope and whether the statements affect a single field, all fields in a single line, or all lines.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:formal-name "Version";
            type string;
            description "Version";
         }
         junos-odl:format xyz-information-format {
            junos-odl:line {
               junos-odl:field "version" {
                   junos-odl:colon;
                   junos-odl:leading;
               }
            }
         }
      }
   }
}
```

When you execute the RPC, the output for that field includes the label and a colon.

```
Version: value
```

comma

The `comma` statement appends a comma to the node's value in the output. It is used with the `space` statement to create comma-delimited fields in a line of output. For example:

```
value1, Label2: value2, value3
```

To generate a comma and a space after a node's value in the output field, include the `comma` and `space` statements within the `format` statement. The placement of the statements within the `format` statement determines the scope. Placing the statements within a single field generates a comma and space for that field only. Placing the statements directly under the `format` statement applies the formatting to all fields.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         // additional leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:comma;
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

If you omit the `comma` statement in the formatting instructions in this example, the fields are separated using only a space. Junos OS automatically omits the comma and space after the last field in a line of output.

default-text

The `default-text` statement specifies the text to display in the formatted ASCII output when the node corresponding to an output field is missing.

To define the string to display when the node mapped to a field is missing, include the `default-text` statement and string within the `field` statement for that node.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf my-model {
            type string;
            description "Model";
         }
```

```
          junos-odl:format xyz-information-format {
             junos-odl:line {
                junos-odl:field "my-model" {
                   junos-odl:default-text "Model number not available.";
                }
             }
          }
       }
    }
}
```

When the node is missing in the RPC reply, the CLI output displays the default text.

```
Model number not available.
```

NOTE: The device only displays the default text when the node is missing. It does not display the text for nodes that are present but empty.

explicit

The `explicit` statement is only used in Junos OS RPCs. You cannot include this statement in custom RPCs.

field and line

The `line` and `field` statements define lines in the RPC's formatted ASCII output and the fields within those lines. You can also use these statements with the `picture` statement to create a more structured table that defines strict column widths and text justification.

To define a line in the formatted ASCII output, include the `line` statement within the `format` statement. Within the `line` statement, include `field` statements that map the leaf nodes in the output tree to fields in the line. The `field` statement's argument is the leaf identifier. Fields must be emitted in the same order as you defined the leaf statements.

The CLI output for the following RPC is a single line with three values. Note that you can include other ODL statements within the `field` and `line` statements to customize the formatting for either a single field or all fields within that line, respectively.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf my-version {
            type string;
            description "Version";
         }
         leaf my-model {
            type string;
            description "Model";
         }
         leaf comment {
            type string;
            description "Comment";
         }
         junos-odl:format xyz-information-format {
            junos-odl:comma;
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "my-version" {
                  junos-odl:capitalize;
               }
               junos-odl:field "my-model";
               junos-odl:field "comment";
            }
         }
      }
   }
}
```

fieldwrap and wordwrap

The `fieldwrap` and `wordwrap` statements enable you to more logically wrap content when a line's width is greater than the width of the display. By default, content that extends past the edge of the display wraps at the point where it meets the right margin, without concern for word boundaries.

The `fieldwrap` statement wraps a field's complete contents to the next line when the current line is so long that it extends past the right edge of the display. If you do not use this statement, the string wraps automatically but without regard for appropriate word breaks or the prevailing margin.

Consider the following lines of output:

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0, Drops: 0, Aged packets: 0
```

If the display is narrower than usual, the line could wrap in the middle of a word as shown in the following sample output:

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0, Dro
ps: 0, Aged packets: 0
```

When you include the `fieldwrap` statement for a field, the entire field is moved to the next line.

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0,
Drops: 0, Aged packets: 0
```

You should only use the `wordwrap` statement on the rightmost column in a table to wrap sections of a multiword value to subsequent lines when the current line is too long. This statement effectively creates a column of text. In the following example, the `wordwrap` statement divides the description string at word boundaries:

```


Packet type     Total          Last 5 seconds    Description
            Sent   Received    Sent   Received
 Hello         0          0       4          5   Establish and maintain
                                                 neighbor relationships.
   DbD        20         25       0          0   (Database description packets)
                                                 Describe the contents of
                                                 the topological database.
 LSReq         6          5       0          0   (Link-State Request packets)
                                                 Request a precise instance
                                                 of the database.
```

To improve the wrapping behavior in the RPC's formatted ASCII output, include the `fieldwrap` statement in each field's formatting instructions. To wrap the rightmost column in a table, include the `wordwrap` statement in the rightmost field's formatting instructions.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
```

```
        }
        leaf desc {
           type string;
           description "Description";
        }
        junos-odl:format xyz-information-format {
           junos-odl:picture "@<<<<<<<<<<<<<<@<<<<<<<<<<<<<<<";
           junos-odl:line {
              junos-odl:field "version" {
                  junos-odl:fieldwrap;
              }
              junos-odl:field "desc" {
                  junos-odl:wordwrap;
              }
           }
        }
     }
   }
}
```

float, header, picture, and truncate

You can create tables in the RPC's formatted ASCII output by defining a `header` statement, a `picture` statement, and one or more `line` statements. The optional `header` statement defines the column headings for a table, but it can also just define general text. The `picture` statement graphically depicts the placement, justification, and width of the columns in a table. The `line` and `field` statements define the table rows and their fields.

The argument for the `picture` statement is a string that includes the at (@), less than (<), greater than (>), and vertical bar (|) symbols to define the placement, justification, and width of the table columns. The @ symbol defines the leftmost position in a column that a value in a field can occupy. The <, >, and | symbols indicate left, right, and center justification, respectively. Repeating the <, >, or | symbol defines the column width. Table 33 on page 603 summarizes the symbols. You can also insert one or more blank spaces between columns.

Table 33: picture Statement Symbols

Symbol	Description
@	Defines the leftmost position in a column that a value in a field can occupy.

Table 33: picture Statement Symbols *(Continued)*

Symbol	Description
|	Centers the contents of the field. Repeated symbols define the column width.
<	Left justifies the contents of the field. Repeated symbols define the column width.
>	Right justifies the contents of the field. Repeated symbols define the column width.

The following `picture` statement defines a left-justified column, a centered column, and a right-justified column that are each six characters wide and separated by a single space:

```
junos-odl:picture "    @<<<<< @||||| @>>>>>";
```

To define a table row, include the `line` statement, and map leaf nodes to fields in the line. The `field` statement's argument is the leaf identifier.

```
            junos-odl:line {
               junos-odl:field "slot";
               junos-odl:field "state";
               junos-odl:field "comment";
            }
```

Sometimes a table field must include one of several mutually exclusive values. In the `picture` statement, you can repeat the @ symbol for each potential value. Then you include the `float` statement within the `field` statement for each mutually exclusive value after the first value. Then if the first element does not have a value, subsequent possible elements with the `float` statement are tested until a value is returned. The value floats into the position defined by the first @ symbol instead of leaving a blank field.

For example, the following `picture` statement causes the output to include one of two mutually exclusive values in the second column:

```
            junos-odl:picture "    @<<<<<  @@<<<<<";
            junos-odl:line {
               junos-odl:field "slot";
               junos-odl:field "state";
               junos-odl:field "comment"{
                  junos-odl:float;
            }
         }
```

You can also use the `float` statement when you know a tag corresponding to a specific table field might be missing in certain situations, and you want to eliminate the extra blank space.

The `truncate` statement guarantees that a field's value does not exceed the width of the column defined by the `picture` statement. The `truncate` statement causes the output to omit any characters in the node's value that would cause it to exceed the width of the field. If you omit the `truncate` statement, and the output exceeds the width of the field, the complete contents are displayed, which might distort the table. You should use this statement with care, particularly with numbers, because the output does not indicate when it truncates a value.

The CLI formatting for the following RPC defines a small table with two columns. The `comment` field includes the `float` and `truncate` statements. If the `state` output element contains a value, the output places the value in the second column. However, if the `state` output element is empty, the output places the value for the `comment` node, if one exists, in the table's second column. If the comment exceeds the width of that column, the output truncates it to fit the column width.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf slot {
            type string;
            description "Slot number";
         }
         leaf state {
            type string;
            description "State";
         }
         leaf comment {
            type string;
         }
         junos-odl:format xyz-information-format {
            junos-odl:header "Slot   State       \n";
            junos-odl:picture "@<<<<< @@|||||||||||||||||||";
            junos-odl:line {
               junos-odl:field "slot";
               junos-odl:field "state";
               junos-odl:field "comment"{
                  junos-odl:float;
                  junos-odl:truncate;
               }
```

```
                }
            }
        }
    }
}
```

format

When you execute an RPC on a device running Junos OS or a device running Junos OS Evolved, it returns the RPC reply as an XML document. Container and leaf nodes under the RPC `output` statement translate into XML elements in the RPC reply. In YANG RPCs for Junos devices, you can also define custom formatted ASCII output. The device displays the formatted ASCII output when you execute the RPC in the CLI or when you request RPC output in text format.

To create custom command output for a specific RPC output container, define the `format` statement. The `format` statement defines the CLI formatting for the parent container. The RPC reply does not include it as a node in the XML data. Within the `format` statement, map the data for the parent container to output fields, and use statements from the Junos OS ODL extensions module to specify how to display the output for that parent container.

```
rpc get-xyz-information {
    output {
        container xyz-information {
            // leaf definitions
            junos-odl:format xyz-information-format {
                // CLI formatting for the parent container
            }
        }
    }
}
```

header and header-group

The `header` statement enables you to define a header string that precedes a set of fields in the RPC's formatted ASCII output. The `header-group` statement emits only the first header string when the output would include two or more headers in the same header group.

To define a header string and associate it with a header group, include the `header` and `header-group` statements, respectively, within the `format` statement. The `header-group` argument is a user-defined string that identifies a particular header group. Every `format` statement that includes the `header-group` statement with the same identifier belongs to the same header group. The following example defines a `format` statement associated with the header group `color-tags`.

```
junos-odl:format red-format {
   junos-odl:header-group "color-tags";
   junos-odl:header "Color tags\n";
   ...
}
```

When multiple `format` statements are associated with the same header group, and the tags emitted by two or more of those statements are present in the output, the CLI output only emits the first header it encounters. The output suppresses any subsequent headers belonging to that header group.

To emit only the first header string for a header group, include the `header-group` statement and identifier in all `format` statements belonging to that header group. The following sample RPC `output` statement associates two containers and their format statements with the header group `color-tags`.

```
output {
   container red-group {
      container red {
         leaf redtag1 {
            type string;
         }
         leaf redtag2 {
            type string;
         }
         junos-odl:format red-format {
            junos-odl:header-group "color-tags";
            junos-odl:header "Color tags\n";
            junos-odl:picture "@<<<<<<<<<<<<  @<<<<<<<<<<<<";
            junos-odl:indent 5;
            junos-odl:line {
               junos-odl:field "redtag1";
               junos-odl:field "redtag2";
            }
         }
      }
   }
   container blue-group {
      container blue {
         leaf bluetag1 {
            type string;
         }
         leaf bluetag2 {
            type string;
         }
         junos-odl:format blue-format {
            junos-odl:header-group "color-tags";
            junos-odl:header "Color tags\n";
            junos-odl:picture "@<<<<<<<<<<<<  @<<<<<<<<<<<<";
            junos-odl:indent 5;
            junos-odl:line {
               junos-odl:field "bluetag1";
               junos-odl:field "bluetag2";
            }
         }
      }
   }
}
```

Consider an RPC reply with the following XML tags:

```
<rpc-reply>
    <red-group>
        <red>
          <redtag1>red-1</redtag1>
          <redtag2>red-2</redtag2>
        </red>
    </red-group>
    <blue-group>
        <blue>
          <bluetag1>blue-1</bluetag1>
          <bluetag2>blue-2</bluetag2>
        </blue>
    </blue-group>
</rpc-reply>
```

When the device renders the output and the same `header-group` statement is present in each `format` statement, it emits only the first header string in the output. In this case, the `format` statement with the identifier `red-format` defines the first header string.

```
Color tags
      red-1          red-2
      blue-1         blue-2
```

If you omit the `header-group` statement from the `format` statement, the output includes the header string defined for each set of fields.

```
Color tags
      red-1          red-2
Color tags
      blue-1         blue-2
```

indent

The `indent` statement causes all of the lines in the scope of the statement other than the header row to be indented by the specified number of characters.

To indent lines, include the `indent` statement and the number of spaces to indent the lines at the top level of the `format` statement. The formatted ASCII output for the following RPC indents the line by 10 spaces.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         leaf model {
            type string;
            description "Model";
         }
         junos-odl:format xyz-information-format {
            junos-odl:header "xyz information\n";
            junos-odl:indent 10;
            junos-odl:line {
               junos-odl:field "version";
               junos-odl:field "model";
            }
         }
      }
   }
}
```

When you execute the RPC, the output left justifies the header and indents the line containing the two fields by ten spaces.

```
xyz information
          *version model*
```

no-line-break

The `no-line-break` statement displays multiple values on the same line in the case where the output emits multiple entities with the same tag names. When you include the `no-line-break` statement, the output places repeated formats on the same line. If you omit the statement, the output places repeated formats on separate lines.

For example, you might want to display all interface errors together on the same line.

```
Interface errors:
BPI-B1 0 BIP-B2 0 REI-L 0 BIP-B3 0 REI-P 0
```

To place the tags for multiple entities within the same line of output, include the `no-line-break` statement in the `format` statement for that container.

```
rpc get-if-errors {
   output {
      container if-error-information {
         container if-errors {
            leaf if-error-name {
               type string;
               description "Interface error name";
            }
            leaf if-error-count {
               type integer;
               description "Interface error count";
            }
            junos-odl:format if-errors-format {
               junos-odl:no-line-break;
```

```
                junos-odl:space;
                junos-odl:header "Interface errors:\n";
                junos-odl:line {
                   junos-odl:field "if-error-name";
                   junos-odl:field "if-error-count";
                }
             }
          }
       }
    }
 }
```

If the RPC output returns multiple entities, the output places each repeated set of fields on the same line.

```
Interface errors:
BPI-B1 0 BIP-B2 0 REI-L 0 BIP-B3 0 REI-P 0
```

If you omit the `no-line-break` statement, the output places each repeated set of fields on its own line.

```
Interface errors:
BPI-B1 0
BIP-B2 0
REI-L 0
BIP-B3 0
REI-P 0
```

space

The `space` statement appends a space to the node's value in the RPC's formatted ASCII output. For example:

```
value1 value2 Label3: value3
```

The `space` statement is often used with the `comma` statement to delimit fields in a line of output with a comma followed by a space.

To generate a space after a value in the output field, include the `space` statement within the `format` statement. The placement of a statement determines the statement's scope. Placing the statement within a single field generates a space after that field only.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         // additional leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

style

The `style` statement defines one of several formats for the RPC output. For detailed information about using the `style` statement to create different levels of output, see "Define Different Levels of Output in Custom YANG RPCs for Junos Devices" on page 614.

template

The `template` statement explicitly defines the format for an output field for a given node. The definition includes the output string and placement of the node's value within the string. If you define the `template` statement for a leaf node, the corresponding output field automatically uses the template string.

To create a template string for a node, you must include the `template` statement in the definition of the node, and define the string. The placeholders `%s` and `%d` within the string define the type and placement of the node's value. Use `%s` to insert a string value, and `%d` to insert an integer value. For example:

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:template " Version: %s";
            type string;
            description "Version";
         }
      }
   }
}
```

If you define a `template` statement for a node, the output field for that node automatically uses the template text.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:template " Version: %s";
            type string;
            description "Version";
         }
         junos-odl:format xyz-information-format {
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

When you execute the RPC, the template is used in the output for that field.

```
 Version: value
```

NOTE: If a leaf statement defines both a `template` and a `formal-name` statement, and the `leading` statement is included in the formatting instructions for that field, the output uses the text defined for the `formal-name` statement and not the text defined for the `template` statement.

RELATED DOCUMENTATION

Understanding the YANG Modules for Junos Operational Commands | 484

Create Custom RPCs in YANG for Devices Running Junos OS | 549

Define Different Levels of Output in Custom YANG RPCs for Junos Devices | 614

Understanding Junos OS YANG Extensions for Formatting RPC Output | 590

Define Different Levels of Output in Custom YANG RPCs for Junos Devices

IN THIS SECTION

Defining Different Levels of Output in Custom YANG RPCs

You can define custom RPCs for Junos devices using YANG. You can customize the RPC output to emit different data and CLI formatting depending on the RPC input. Thus, you can create different styles, or levels of output, for the same RPC.

You can request the output style by including the appropriate value for the input argument when you invoke the RPC. The action script must process this argument and emit the XML output for the requested style. Junos OS then translates the XML into the corresponding CLI output defined for that style in the YANG module. The RPC template presented in this topic creates two styles: `brief` and `detail`.

To create different styles for the output of an RPC:

1. In the YANG module that includes the RPC, import the Junos OS ODL extensions module. The module defines YANG extensions that you use to precisely specify how to render the text output.

```
import junos-extension-odl {
   prefix junos-odl;
}
```

2. In the RPC's input parameters, define a `leaf` statement with type `enumeration`. Include `enum` statements that define names for each style.

```
rpc rpc-name {
   description "RPC description";
   junos:command "cli-command" {
      junos:action-execute {
         junos:script "action-script-filename";
      }
   }

   input {
      leaf level {
         type enumeration {
            enum brief {
               description "Display brief output";
            }
            enum detail {
               description "Display detailed output";
            }
         }
      }
   }
}
```

3. In the RPC `output` statement, create separate `junos-odl:style` statements that define the CLI formatting for each style. The identifier for each `style` statement should match one of the style names defined within the enumerated `leaf` statement.

```
   output {
      container output-container {

         //  leaf definitions

         junos-odl:style brief {
            junos-odl:format output-container-format-brief {
               // formatting for brief output
            }
         }
         junos-odl:style detail {
            junos-odl:format output-container-format-detail {
               // formatting for detailed output
            }
         }

      }
   }
```

4. In the RPC's action script, first process the input argument. Then emit the XML output for the requested style enclosed in a parent element that has a tag name identical to the style name.

   NOTE: Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--level', required=False, default='brief')
parser.add_argument('--rpc_name', required=True)
args = parser.parse_args()

print ("<output-container>")
print ("<{}>".format(args.level))     # tag name is brief or detail

if args.level == "brief":
   # print statements for brief output

if args.level == "detail":
  # print statements for detailed output

print ("</{}>".format(args.level))
print ("</output-container>")
```

See "Example: Defining Different Levels of Output" on page 618 for full script examples that work in the various releases.

The following code outlines the general structure of the RPC and enclosing module. When you invoke the RPC in the CLI, include the input argument `level`, and specify either `brief` or `detail`, Junos OS renders the output defined for that style.

```
module module-name {
   namespace "http://yang.juniper.net/yang/1.1/jrpc";
   prefix jrpc;

   import junos-extension {
      prefix junos;
   }
   import junos-extension-odl {
      prefix junos-odl;
   }

   organization
      "Juniper Networks, Inc.";
   description
      "Junos OS YANG module for custom RPCs";

   rpc rpc-name {
      description "RPC description";

      junos:command "cli-command" {
         junos:action-execute {
            junos:script "action-script-filename";
         }
      }

      input {
         leaf level {
            type enumeration {
               enum brief {
                  description "Display brief output";
               }
               enum detail {
                  description "Display detailed output";
               }
            }
         }
      }
      output {
         container output-container {

            // leaf definitions

            junos-odl:style brief {
               junos-odl:format output-container-format-brief {
                  // formatting for brief output
               }
            }

            junos-odl:style detail {
               junos-odl:format output-container-format-detail {
                  // formatting for detailed output
               }
            }

         }
      }
   }
}
```

To execute the RPC in the CLI, issue the command defined by the `junos:command` statement. Specify the style by including the appropriate command-line argument, which in this example is `level`. The corresponding action script processes the input argument and emits the output for the requested style.

```
user@host> cli-command level brief
```

Example: Defining Different Levels of Output

IN THIS SECTION

This example presents a simple custom YANG RPC and action script that determine if a host is reachable and print different levels of output depending on the user input.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS or device running Junos OS Evolved that supports loading custom YANG data models.

Overview of the RPC and Action Script

The YANG module in this example defines a custom RPC to ping the specified host and return the result using different levels of output based on the user's input. The YANG module `rpc-style-test` is saved in the **rpc-style-test.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the CLI output.

The module defines the `get-host-status` RPC. The `<get-host-status>` request tag is used to remotely execute the RPC on the device. In the RPC definition, the `junos:command` statement defines the command that you use to execute the RPC in the CLI, which in this case is `show host-status`.

```
rpc get-host-status {
   description "RPC example to retrieve host status";

   junos:command "show host-status" {
      junos:action-execute {
         junos:script "rpc-style-test.py";
      }
   }
  ...
```

The `junos:action-execute` and `junos:script` statements define the action script that is invoked when you execute the RPC. This example uses a Python action script named **rpc-style-test.py** to retrieve the information required by the RPC. The script returns the XML output elements for each level of output as defined in the RPC's `output` statement.

The RPC has two input parameters, `hostip` and `level`. The `hostip` parameter is the host to check for reachability. The `level` parameter selects the style for the RPC's output. When you execute the RPC, you include the target host's IP address and a level, `brief` or `detail`. The action script defines the default value for `level` as `'brief'`, so if you omit this argument, the RPC prints the output corresponding to the brief style.

```
input {
   leaf hostip {
      description "Host IP address";
      type string;
   }
   leaf level {
      type enumeration {
         enum brief {
            description "Display brief output";
         }
         enum detail {
            description "Display detailed output";
         }
      }
   }
}
```

The RPC also defines the output nodes that must be emitted by the corresponding action script. The root node is the `<host-status-information>` element. This root node encloses either the `<brief>` or the `<detail>` element, depending on the user input. It also includes the child output nodes specified for each level of output. Both levels of output include the `<hostip>` and `<status>` child elements, but the `<detail>` element also includes the `<date>` child element. The `junos-odl:format` statements define the formatting for the text output. The output XML tree does not include this node.

```
output {
   container host-status-information {
      ...
      junos-odl:style brief {
         junos-odl:format host-status-information-format-brief {
             ...
         }
      }
      junos-odl:style detail {
         junos-odl:format host-status-information-format-detail {
            ...
         }
      }
```

```
        }
    }
```

The action script pings the host to determine if it is reachable and sets the status based on the results. The script then constructs and prints the XML for the RPC output based on the specified `level` argument. The XML tree must exactly match the hierarchy defined in the RPC.

You add the module containing the RPC and the action script file to the device as part of a new YANG package named `rpc-style-test`.

YANG Module and Action Script

IN THIS SECTION

YANG Module

The YANG module, **rpc-style-test.yang**, defines the RPC, the command used to execute the RPC in the CLI, and the name of the action script to invoke when you execute the RPC. The base name of the file must match the module name.

```
/*
* Copyright (c) 2024 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-style-test {
  namespace "http://yang.juniper.net/yang/1.1/jrpc";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }
```

```
organization
  "Juniper Networks, Inc.";

description
  "Junos OS YANG module for RPC example";

rpc get-host-status {
   description "RPC example to retrieve host status";

   junos:command "show host-status" {
      junos:action-execute {
         junos:script "rpc-style-test.py";
      }
   }

   input {
      leaf hostip {
         description "Host IP address";
         type string;
      }
      leaf level {
         type enumeration {
            enum brief {
               description "Display brief output";
            }
            enum detail {
               description "Display detailed output";
            }
         }
      }
   }
   output {
      container host-status-information {
         leaf hostip {
            type string;
            description "Host IP";
         }
         leaf status {
            type string;
            description "Operational status";
         }
         leaf date {
            type string;
```

```
              description "Date information";
           }
           junos-odl:style brief {
              junos-odl:format host-status-information-format-brief {
                 junos-odl:header "Brief output\n";
                 junos-odl:picture "@<<<<<<<<<<<< @";
                 junos-odl:space;
                 junos-odl:line {
                   junos-odl:field "hostip";
                   junos-odl:field "status";
                 }
              }
           }
           junos-odl:style detail {
              junos-odl:format host-status-information-format-detail {
                 junos-odl:header "Detail output\n";
                 junos-odl:picture "@<<<<<<<<<<<<  @<<<<<<<<<<<< @";
                 junos-odl:space;
                 junos-odl:line {
                    junos-odl:field "hostip";
                    junos-odl:field "status";
                    junos-odl:field "date";
                 }
              }
           }
        }
     }
  }
}
```

Action Script

The corresponding action script is **rpc-style-test.py**. The action script prints different levels of output based on the value of the `level` argument provided by the user. The script defines a default value of `'brief'` for the `level` argument so that if the user omits the argument, the script returns the brief style of output. This example provides two versions of the action script, which appropriately handle the script's command-line arguments for the different releases.

Action Script (Junos OS Release 21.2R1 and later)

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import os
import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--hostip', required=True)
parser.add_argument('--level', required=False, default='brief')
parser.add_argument('--rpc_name', required=True)
args = parser.parse_args()

f = os.popen('date')
now = f.read()

# Ping target host and set the status
response = os.system('ping -c 1 ' + args.hostip + ' > /dev/null')
if response == 0:
    pingstatus = "Host is Active"
else:
    pingstatus = "Host is Inactive"

# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args.level))
print ("<hostip>{}</hostip>".format(args.hostip))
print ("<status>{}</status>".format(pingstatus))
if args.level == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args.level))
print ("</host-status-information>")
```

Action Script (Junos OS Release 21.1 and earlier)

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
import os

args = {'hostip': None, 'level': 'brief'}

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]

f = os.popen('date')
now = f.read()

# Ping target host and set the status
if args['hostip'] is not None:
    response = os.system('ping -c 1 ' + args['hostip'] + ' > /dev/null')
    if response == 0:
        pingstatus = "Host is Active"
    else:
        pingstatus = "Host is Inactive"
else:
    pingstatus = "Invalid host"

# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args['level']))
print ("<hostip>{}</hostip>".format(args['hostip']))
print ("<status>{}</status>".format(pingstatus))
if args['level'] == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args['level']))
print ("</host-status-information>")
```

Configuration

IN THIS SECTION

Enable Execution of Python Scripts

To enable the device to execute unsigned Python scripts:

1. Configure the `language python` or `language python3` statement, as appropriate for the Junos OS release.

```
[edit]
user@host# set system scripts language (python | python3)
```

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

2. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

Load the RPC on the Device

To add the RPC and action script to the Junos schema on the device:

1. Download the YANG module and action script to the Junos device.
2. Ensure that the Python action script meets the following requirements:
   - File owner is either root or a user in the Junos OS `super-user` login class.
   - Only the file owner has write permission for the file.
   - Script includes the appropriate interpreter directive line as outlined in "Create Action Scripts for YANG RPCs on Junos Devices" on page 556.
3. (Optional) Validate the syntax for the YANG module and action script.

```
user@host> request system yang validate module /var/tmp/rpc-style-test.yang action-
script /var/tmp/rpc-style-test.py
YANG modules validation : START
YANG modules validation : SUCCESS
```

```
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

4. Add the YANG module and action script to a new YANG package.

```
user@host> request system yang add package rpc-style-test module /var/tmp/rpc-style-test.yang
action-script /var/tmp/rpc-style-test.py
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Restarting mgd ...
```

5. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`, or type **yes** and press `Enter`.

```
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

Verify the RPC

IN THIS SECTION

Purpose

Verify that the RPC works as expected.

Action

From operational mode, execute the RPC in the CLI by issuing the command defined by the `junos:command` statement in the RPC definition. Include the `hostip` input argument, and include the `level` argument for each different level of output.

```
user@host> show host-status hostip 198.51.100.1 level brief
Brief output
198.51.100.1   Host is Active
```

You can view the corresponding XML by appending `| display xml` to the command.

```
user@host> show host-status hostip 198.51.100.1 level brief | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
    <host-status-information>
        <brief>
            <hostip>
                198.51.100.1
            </hostip>
            <status>
                Host is Active
            </status>
        </brief>
    </host-status-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Similarly, for the detailed output:

```
user@host> show host-status hostip 198.51.100.10 level detail
Detail output
198.51.100.10   Host is Inactive Fri Feb  8 11:55:54 PST 2019
```

```
user@host> show host-status hostip 198.51.100.10 level detail | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
    <host-status-information>
        <detail>
```

```
                <hostip>
                    198.51.100.10
                </hostip>
                <status>
                    Host is Inactive
                </status>
                <date>
                    Fri Feb  8 16:03:35 PST 2019
                </date>
            </detail>
        </host-status-information>
        <cli>
            <banner></banner>
        </cli>
    </rpc-reply>
```

Meaning

When you execute the RPC, the device invokes the action script. The action script prints the XML hierarchy for the given level of output as defined in the RPC `output` statement. When you execute the RPC in the CLI, the device uses the CLI formatting defined in the RPC to convert the XML output into the displayed CLI output.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1 and 21.2R1-EVO	Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

RELATED DOCUMENTATION

Create Custom RPCs in YANG for Devices Running Junos OS | **549**

Understanding Junos OS YANG Extensions for Formatting RPC Output | **590**

Customize YANG RPC Output on Devices Running Junos OS | **594**

Display Valid Command Option and Configuration Statement Values in the CLI for Custom YANG Modules

IN THIS SECTION

Certain Junos devices enable you to load custom YANG modules on the device to add data models that are not natively supported by Junos OS. When you add custom YANG data models to a device, you must also supply an action or translation script that handles the translation logic between the YANG data model and Junos OS. Although the script logic can ensure that a user supplies valid values for a given command option or configuration statement, that logic is not always transparent to the user. Starting in Junos OS Release 19.2R1, the CLI displays the set of possible values for certain command options or configuration statements in a custom YANG data model when you include the `action-expand` extension statement in the option or statement definition and reference a script that handles the logic.

Understanding Context-Sensitive Help for Custom YANG Modules

The Junos CLI provides context-sensitive help whenever you type a question mark (?) in operational or configuration mode. When you execute a command or configure a statement, the CLI's context-sensitive help displays the valid options and option values for a command or the valid configuration statements and leaf statement values in the configuration statement hierarchy. Additionally, context-sensitive help shows the possible completions for incomplete option names, statement names, and their values.

The CLI can also display the values that are valid for certain command options or configuration statements in a custom YANG data model. The CLI can display all possible values or a subset of values that match on partial input from the user. For example:

```
user@host> show host-status hostip ?
Possible completions:
  <hostip>             Host IP address
  10.10.10.1           IPv4 address
  10.10.10.2           IPv4 address
  172.16.0.1           IPv4 address
  198.51.100.1         IPv4 address
  198.51.100.10        IPv4 address
  2001:db8::1          IPv6 address (DC 1...128)
  2001:db8::fdd2       IPv6 address (DC 1...128)
```

```
user@host> show host-status hostip 198?
Possible completions:
  <hostip>             Host IP address
  198.51.100.1         IPv4 address
  198.51.100.10        IPv4 address
```

To display the set of valid values for a given command option or configuration statement in a custom YANG module:

1. Define the `action-expand` and `script` extension statements under the appropriate input parameter or configuration statement in the YANG module as described in "Defining the YANG Module" on page 631.

2. Create a Python script that checks for user input, calculates the possible values of the command option or configuration statement, and sends the appropriate output to the CLI, as described in "Creating the CLI Expansion Script" on page 633.

   NOTE: The CLI expansion script only displays the valid values in the CLI. The module's translation script or action script must still include the logic that ensures that only valid values are accepted and processed.

3. Load the YANG module, any translation or action scripts, and the CLI expansion script as part of a custom YANG package on the device as described in "Loading the YANG Package" on page 636.

   NOTE: Junos devices process CLI expansion scripts as another kind of action script, but we refer to CLI expansion script to avoid any confusion.

Defining the YANG Module

To define a custom YANG module that displays the set of valid values for a given command option or configuration statement when the user requests context-sensitive help in the CLI, your module must:

1. Import the Junos OS DDL extensions module.

2. Include the `action-expand` extension statement and `script` substatement in the corresponding command option or configuration statement definition.

   - You can include the `action-expand` statement within a `leaf` statement in modules that define custom RPCs and within a `leaf` or `leaf-list` statement in modules that define custom configuration hierarchies.

   - You can only define a single `action-expand` statement for a given node.

   - The `script` statement should reference the Python script that defines your custom logic.

For example, in the following module, the RPC defines the `hostip` input parameter, which calls the `hostip-expand.py` Python script when the user requests context-sensitive help for the `hostip` argument in the CLI. The script implements the custom logic that displays the valid values for that argument in the CLI.

```
module rpc-host-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  rpc get-host-status {
     description "RPC example to retrieve host status";

     junos:command "show host-status" {
        junos:action-execute {
           junos:script "rpc-host-status.py";
        }
     }

     input {
        leaf hostip {
           description "Host IP address";
           type string;
           junos:action-expand {
              junos:script "hostip-expand.py";
           }
        }
        leaf level {
           type enumeration {
              enum brief {
                 description "Display brief output";
              }
              enum detail {
                 description "Display detailed output";
              }
           }
        }
     }
     output {
     ...
     }
  }
}
```

Creating the CLI Expansion Script

When you define the `action-expand` statement and `script` substatement for a command option or configuration statement in a custom YANG module and you request context-sensitive help for that option or statement value in the CLI, the device invokes the referenced Python script. The script must contain the custom logic that calculates and displays all possible values for that parameter or displays a subset of values that match on partial input from the user.

For example, the following command should display all valid values for the `hostip` argument:

```
user@host> show host-status hostip ?
```

And the following command should display all valid values that start with "198":

```
user@host> show host-status hostip 198?
```

To display the valid values for a command option or configuration statement in the CLI, the Python script should perform the following functions:

1. Import the `jcs` library along with any other required Python libraries.

2. Retrieve and process any user input.

   If you specify partial input for an option or statement value in the CLI, the script's command-line arguments include the `symbol` argument, which is a string containing the user input.

NOTE: Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, the script's command-line arguments include the `--symbol` argument instead of the `symbol` argument.

3. Define or calculate the valid values for the parameter.

4. Call the `jcs.expand()` function for each value to display on the command line.

The script must call the `jcs:expand()` function for each option or statement value to display in the CLI. The syntax for the `jcs:expand()` function is:

```
jcs.expand(value, description, <units>, <range>)
```

Where:

value	String defining a valid value for the given command option or configuration statement.
description	String that describes the value.
units	(Optional) String that defines the units for the corresponding value.
range	(Optional) String that defines the range for the corresponding value.

For each call to the `jcs.expand()` function, the script emits the value, description, units, and range that are provided in the function arguments in the CLI. For example, given the following call to `jcs.expand()` in the script:

```
jcs.expand("2001:db8:4136::fdd2", "IPv6 address", "DC", "1...128")
```

The corresponding CLI output is:

```
Possible completions:
  <hostip>             Host IP address
  2001:db8:4136::fdd2   IPv6 address (DC 1...128)
```

The following sample scripts first check for the presence of `symbol` in the script's command-line arguments, and if present, set the corresponding variable equal to the user's input. The scripts then calculate the set of valid values for the parameter based on the user's input. Finally, the scripts call the `jcs.expand()` function for each value to display in the CLI.

We provide two versions of the script, which appropriately handle the script's `symbol` argument for the different releases. The following sample script, which is valid on devices running Junos OS Release 21.2R1 or later, uses the `argparse` library to parse the `--symbol` argument.

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import jcs
import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--symbol', required=False, default='')
args = parser.parse_args()

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

for ip in item:
    if ip.startswith(args.symbol) or not args.symbol:
        if not expand_colon in ip:
            jcs.expand(ip, description_ipv4)
        else:
            jcs.expand(ip, description_ipv6,
            expand_units, expand_range)
```

Similarly, the following sample script, which is valid on devices running Junos OS Release 21.1 or earlier, checks for `symbol` in the `sys.argv` list.

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
import jcs

symbol = ""
```

```
# Retrieve user input in symbol argument and store the value
if "symbol" in sys.argv:
    index = sys.argv.index("symbol")
    symbol = sys.argv[index+1]

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

for ip in item:
    if ip.startswith(symbol) or not symbol:
        if not expand_colon in ip:
            jcs.expand(ip, description_ipv4)
        else:
            jcs.expand(ip, description_ipv6,
            expand_units, expand_range)
```

The CLI expansion script only displays the valid values, units, and ranges for the command option or configuration statement in the CLI. The module's translation script or action script must ensure that only valid values are accepted and processed.

Loading the YANG Package

When you load a YANG package on a Junos device, include any CLI expansion scripts in the list of action scripts for that package. Junos OS automatically copies the script to the **/var/db/scripts/action** directory.

To load a new package and include custom CLI expansion scripts:

1. Ensure that the Python scripts meet the following requirements:
    - File owner is either root or a user in the Junos OS `super-user` login class.
    - Only the file owner has write permission for the file.
    - Script includes an interpreter directive line as outlined in "Create Action Scripts for YANG RPCs on Junos Devices" on page 556.

2. In configuration mode, enable the device to execute unsigned Python scripts by configuring the `language python` or `language python3` statement, as appropriate for the Junos OS release.

```
[edit]
user@host# set system scripts language (python | python3)
user@host# commit and-quit
```

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

3. In operational mode, load the YANG package, and include the CLI expansion script in the `action-script` list.

```
user@host> request system yang add package rpc-host-status module /var/tmp/rpc-host-
status.yang action-script [/var/tmp/rpc-host-status.py /var/tmp/hostip-expand.py]
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...
```

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations, change the configuration, or terminate the operation while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

4. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`.

```
...
WARNING: cli has been replaced by an updated version:
```

```
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

Example: Displaying Context-Sensitive Help for a Command Option

IN THIS SECTION

This example presents a custom YANG module that uses the `action-expand` extension statement and a custom script to display the set of possible values for one of the command options when a user requests context-sensitive help in the CLI for that option.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS Release 19.2R1 or later that supports loading custom YANG data models.

Overview

The YANG module in this example defines a custom RPC to ping the specified host and return the result. The YANG module `rpc-host-status` is saved in the **rpc-host-status.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the output and context-sensitive help in the CLI.

The module defines the `get-host-status` RPC. The `junos:command` statement defines the command that is used to execute the RPC in the CLI, which in this case is `show host-status`. The `junos:action-execute` and `junos:script` statements define the action script that is invoked when you execute the RPC.

```
rpc get-host-status {
   description "RPC example to retrieve host status";
```

```
junos:command "show host-status" {
   junos:action-execute {
      junos:script "rpc-host-status.py";
   }
}
```

The `hostip` input parameter includes the `junos:action-expand` and `junos:script` statements, which define the script that is invoked when the user requests context-sensitive help in the CLI for that input parameter.

```
input {
   leaf hostip {
      description "Host IP address";
      type string;
      junos:action-expand {
         junos:script "hostip-expand.py";
      }
   }
   ...
}
```

The **hostip-expand.py** script processes the user's input, which is passed to the script as the argument `symbol` or `--symbol`, depending on the release. The script then calculates and displays the set of values that the user can enter for that command option.

NOTE: Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script (including CLI expansion scripts), it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

The expansion script displays the valid values for `hostip` in the CLI. The action script implements the logic that determines if the provided value is valid. This example adds the YANG module and the action scripts to the device as part of a new YANG package named `rpc-host-status`.

YANG Module and Action Scripts

IN THIS SECTION

YANG Module

The YANG module, **rpc-host-status.yang**, defines the RPC, the command used to execute the RPC in the CLI, the name of the action script to invoke when you execute the RPC, and the name of the CLI expansion script to invoke when the user requests context-sensitive help for the corresponding input parameter.

```
/*
* Copyright (c) 2019 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-host-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  organization
    "Juniper Networks, Inc.";

  description
    "Junos OS YANG module for RPC example";

  rpc get-host-status {
     description "RPC example to retrieve host status";

     junos:command "show host-status" {
        junos:action-execute {
           junos:script "rpc-host-status.py";
        }
     }

     input {
        leaf hostip {
           description "Host IP address";
           type string;
           junos:action-expand {
              junos:script "hostip-expand.py";
           }
        }
        leaf level {
           type enumeration {
              enum brief {
                 description "Display brief output";
              }
              enum detail {
                 description "Display detailed output";
              }
           }
        }
     }
     output {
        container host-status-information {
           leaf hostip {
              type string;
              description "Host IP";
           }
           leaf status {
              type string;
              description "Operational status";
           }
           leaf date {
              type string;
              description "Date information";
           }
           junos-odl:style brief {
              junos-odl:format host-status-information-format-brief {
                 junos-odl:header "Brief output\n";
                 junos-odl:picture "@<<<<<<<<<<<< @";
                 junos-odl:space;
                 junos-odl:line {
                   junos-odl:field "hostip";
                   junos-odl:field "status";
```

```
                    }
                }
            }
            junos-odl:style detail {
                junos-odl:format host-status-information-format-detail {
                    junos-odl:header "Detail output\n";
                    junos-odl:picture "@<<<<<<<<<<<<<  @<<<<<<<<<<<<< @";
                    junos-odl:space;
                    junos-odl:line {
                        junos-odl:field "hostip";
                        junos-odl:field "status";
                        junos-odl:field "date";
                    }
                }
            }
        }
    }
  }
}
```

Action Script

The corresponding action script is **rpc-host-status.py**. This example provides two versions of the action script, which appropriately handle the script's command-line arguments for the different releases.

Action Script (Junos OS Release 21.2R1 and later)

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import os
import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--hostip', required=True)
parser.add_argument('--level', required=False, default='brief')
parser.add_argument('--rpc_name', required=True)
args = parser.parse_args()

valid_addresses = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]
```

```
f = os.popen('date')
now = f.read()

# Ping target host and set the status
if args.hostip in valid_addresses:
    response = os.system('ping -c 1 ' + args.hostip + ' > /dev/null')
    if response == 0:
        pingstatus = "Host is Active"
    else:
        pingstatus = "Host is Inactive"
else:
    pingstatus = "Invalid host"


# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args.level))
print ("<hostip>{}</hostip>".format(args.hostip))
print ("<status>{}</status>".format(pingstatus))
if args.level == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args.level))
print ("</host-status-information>")
```

Action Script (Junos OS Release 21.1 and earlier)

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
import os

args = {'hostip': None, 'level': 'brief'}
valid_addresses = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]

f = os.popen('date')
now = f.read()

# Ping target host and set the status
if args['hostip'] in valid_addresses:
    response = os.system('ping -c 1 ' + args['hostip'] + ' > /dev/null')
    if response == 0:
        pingstatus = "Host is Active"
    else:
        pingstatus = "Host is Inactive"
else:
    pingstatus = "Invalid host"

# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args['level']))
print ("<hostip>{}</hostip>".format(args['hostip']))
print ("<status>{}</status>".format(pingstatus))
if args['level'] == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args['level']))
print ("</host-status-information>")
```

CLI Expansion Script

The action script that handles the logic to display the valid values for `hostip` in the CLI is **hostip-expand.py**. This example provides two versions of the script, which appropriately handle the script's arguments for the different releases.

CLI expansion script (Junos OS Release 21.2R1 and later)

```
#!/usr/bin/python3
# Junos OS Release 21.2R1 and later

import jcs
import argparse

parser = argparse.ArgumentParser(description='This is a demo script.')
parser.add_argument('--symbol', required=False, default='')
```

```
args = parser.parse_args()

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

for ip in item:
    if ip.startswith(args.symbol) or not args.symbol:
        if not expand_colon in ip:
            jcs.expand(ip, description_ipv4)
        else:
            jcs.expand(ip, description_ipv6,
            expand_units, expand_range)
```

CLI expansion script (Junos OS Release 21.1 and earlier)

```
#!/usr/bin/python
# Junos OS Release 21.1 and earlier

import sys
import jcs

symbol = ""

# Retrieve user input in symbol argument and store the value
if "symbol" in sys.argv:
    index = sys.argv.index("symbol")
    symbol = sys.argv[index+1]

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]


for ip in item:
    if ip.startswith(symbol) or not symbol:
        if not expand_colon in ip:
            jcs.expand(ip, description_ipv4)
        else:
            jcs.expand(ip, description_ipv6,
            expand_units, expand_range)
```

Configuration

IN THIS SECTION

Enable Execution of Python Scripts

To enable the device to execute unsigned Python scripts:

1. Configure the `language python` or `language python3` statement, as appropriate for the Junos OS release.

```
[edit]
user@host# set system scripts language (python | python3)
```

NOTE: Starting in Junos OS Release 20.2R1 and Junos OS Evolved Release 22.3R1, the device uses Python 3 to execute YANG action and translation scripts. In earlier releases, Junos OS only uses Python 2.7 to execute these scripts, and Junos OS Evolved uses Python 2.7 by default to execute the scripts.

2. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

Load the YANG Module and Scripts on the Device

To add the YANG module and scripts to the Junos device:

1. Download the YANG module and scripts to the Junos device.
2. Ensure that the Python scripts meet the following requirements:
   - File owner is either root or a user in the Junos OS `super-user` login class.
   - Only the file owner has write permission for the file.
   - Script includes the appropriate interpreter directive line as outlined in "Create Action Scripts for YANG RPCs on Junos Devices" on page 556.
3. (Optional) Validate the syntax for the YANG module and action scripts.

```
user@host> request system yang validate module /var/tmp/rpc-host-status.yang action-
script [ /var/tmp/rpc-host-status.py /var/tmp/hostip-expand.py ]
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

4. Add the YANG module and scripts to a new YANG package.

```
user@host> request system yang add package rpc-host-status module /var/tmp/rpc-host-
status.yang action-script [ /var/tmp/rpc-host-status.py /var/tmp/hostip-expand.py ]
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
```

```
mgd: commit complete
Restarting mgd ...
```

5. When the system prompts you to restart the Junos OS CLI, press `Enter` to accept the default value of `yes`, or type **yes** and press `Enter`.

```
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

Verifying the Context-Sensitive Help

IN THIS SECTION

Purpose

Verify that the CLI expansion script works as expected.

Action

From operational mode, request context-sensitive help in the CLI by issuing the command defined by the `junos:command` statement in the RPC definition, and include the `hostip` input argument and a question mark (?).

```
user@host> show host-status hostip ?
Possible completions:
  <hostip>             Host IP address
  10.10.10.1           IPv4 address
  10.10.10.2           IPv4 address
  172.16.0.1           IPv4 address
  198.51.100.1        IPv4 address
  198.51.100.10       IPv4 address
  2001:db8::1         IPv6 address (DC 1...128)
  2001:db8::fdd2      IPv6 address (DC 1...128)
```

Perform the same operation with partial user input and verify that the displayed values correctly match the input.

```
user@host> show host-status hostip 198?
Possible completions:
  <hostip>            Host IP address
  198.51.100.1        IPv4 address
  198.51.100.10       IPv4 address
```

Meaning

When context-sensitive help is requested for the `hostip` value, the device invokes the `hostip-expand.py` script. The script processes the user's input, if provided, and prints the valid completions in the CLI. If no user input is given, the script prints all possible values. When user input is provided, the script prints only matching values.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1 and 21.2R1-EVO	Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python action script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

Configure a NETCONF Proxy Telemetry Sensor in Junos

IN THIS SECTION

Using Junos telemetry streaming, you can turn any available state information into a telemetry sensor by means of the XML Proxy functionality. The NETCONF XML management protocol and Junos XML API fully document all options for every supported Junos OS operational request. After you configure XML proxy sensors, you can access data over NETCONF “get” remote procedure calls (RPCs).

This task shows you how to stream the output of a Junos OS operational mode command.

BEST PRACTICE: We recommend not to use YANG files that map to a Junos OS operational command with extensive or verbose output or one that is slow in producing output. Commands with a noticeable delay should be avoided in YANG files. Including such commands can affect other xmlproxyd sensors as well as the performance of xmlproxyd.
The output from some operational mode commands is dynamic and the level of their verbosity depends on factors such as the configuration and hardware. Examples of such commands include any variation of `show interfaces`, `show route`, `show arp`, `show bfd`, `show bgp`, and `show ddos-protection`.

To check the verbosity level of a command, issue the ***command-name***| **display xml** | **count** command. If the line count exceeds a value of 4000 lines, then the command is not recommended for XML proxy streaming. This value is more of an approximation based on internal base-lining. It can be less depending upon various factors such as device type, processing power of the device, and the existing CPU load. Consequently, this feature needs to be used judiciously based on how the device is performing.

You can issue the command ***command-name***| **display xml** before using a YANG file that maps to a Junos OS or Junos OS Evolved operational mode command to verify that the command produces valid XML output and does not contain invalid tags, data, or formatting.

Using a YANG file that maps to a verbose command results in one or more of following:

- The xmlproxyd process CPU utilization remains high. If xmlproxyd has tracing enabled, the CPU utilization is even higher.
- An increase in the xmlproxyd process memory utilization.
- The xmlproxyd process state may show `sbwait`, indicating that the command output is verbose and that xmlproxyd is spending significant time reading the command's remote procedure call's (RPC's) output.
- The xmlproxyd sensor data does not complete the wrap.
- The xmlproxyd streams partial or no data for the sensors.
- The xmlproxyd misses reporting-interval cycles. The intervals start to overlap because of a command's verbose output, resulting in the xmlproxyd's sensor streaming data that is slow or delayed.
- The process or application that serves the verbose command's RPC may show high CPU numbers or delays in performing main tasks. This behavior is caused when the process or application is busy serving the RPC that has verbose output.

This task requires the following:

- An MX Series, vMX Series, or PTX Series router operating Junos OS Release 18.1R1 or later.
- A telemetry data receiver, such as OpenNTI, to verify proper operation of your telemetry sensor.

In this task, you will stream the contents of the Junos OS command `show system users`.

show system users (vMX Series)

```
user@switch> show system users
USER     TTY      FROM                              LOGIN@  IDLE WHAT
user1   pts/0      172.31.12.36                     12:40PM     39 -cli (cli)
user2   pts/1      172,16.03.25                      3:01AM      - -cli (cli)
```

In addition to the expected list of currently logged-in users, the `show system users` output also provides the average system load as 1, 5 and 15 minutes. You can find the load averages by using the `show system users | display xml` command to view the XML tagging for the output fields. See `<load-average-1>`, `<load-average-5>`, and `<load-average-15>` in the XML tagging output below.

```
user@switch> show system users | display xml

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.4R1/junos">
```

```
    <system-users-information xmlns="http://xml.juniper.net/junos/17.4R1/junos">
        <uptime-information>
            <date-time junos:seconds="1520170982">1:43PM</date-time>
            <up-time junos:seconds="86460">1 day, 40 mins</up-time>
            <active-user-count junos:format="2 users">2</active-user-count>
            <load-average-1>0.70</load-average-1>
            <load-average-5>0.58</load-average-5>
            <load-average-15>0.55</load-average-15>
            <user-table>
                <user-entry>
                    <user>root</user>
                    <tty>pts/0</tty>
                    <from>172.21.0.1</from>
                    <login-time junos:seconds="1520167202">12:40PM</login-time>
                    <idle-time junos:seconds="0">-</idle-time>
                    <command>cli</command>
                </user-entry>
                <user-entry>
                    <user>mwiget</user>
                    <tty>pts/1</tty>
                    <from>66.129.241.10</from>
                    <login-time junos:seconds="1520170862">1:41PM</login-time>
                    <idle-time junos:seconds="60">1</idle-time>
                    <command>cli</command>
                </user-entry>
            </user-table>
        </uptime-information>
    </system-users-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

TIP: The `uptime-information` tag shown in the preceding output is a container that contains leafs, such as `date-time`, `up-time`, `active-user-count`. and `load-average-1`. Below is a sample YANG file for this container:

```
container uptime-information {
      dr:source "uptime-information"; // Exact name of the XML tag
      leaf date-time { // YANG model leaf
```

```
        type string; // Type of value
        dr:source date-time; // Exact name of the XML tag
      }
      leaf up-time { // YANG model leaf
        type string; // Type of value
        dr:source up-time; // Exact name of the XML tag
      }
      leaf active-user-count { // YANG model leaf
        type int32; // Type of value
        dr:source active-user-count; // Exact name of the XML tag
      }
      leaf load-average-1 { // YANG model leaf
        type string; // Type of value
        dr:source load-average-1; // Exact name of the XML tag
      }
      ...
```

TIP: The `uptime-information` tag also has another container named `user-table` that contains a list of user entries.

Below is a sample YANG file for this container:

```
container user-table { // "user-table" container which contains list of user-entry
        dr:source "user-table"; // Exact name of the XML tag
        list user-entry { // "user-entry" list which contains the users' details in form of leafs
          key "user"; // Key for the list "user-entry" which is a leaf in the list "user-entry"
          dr:source "user-entry"; // Source of the list "user-entry" which is the exact name of
the XML tag
          leaf user { // YANG model leaf
            dr:source user; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
          leaf tty { // YANG model leaf
            dr:source tty; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
          leaf from { // YANG model leaf
            dr:source from; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
          leaf login-time { // YANG model leaf
```

```
            dr:source login-time; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
          leaf idle-time { // YANG model leaf
            dr:source idle-time; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
          leaf command { // YANG model leaf
            dr:source command; // A leaf in the list "user-entry", exact name of the XML tag
            type string; // Type of value
          }
        }
      }
```

Create a User-Defined YANG File

The YANG file defines the Junos CLI command to be executed, the resource path the sensors are placed under, and the key value pairs taken from the matching XML tags.

Custom YANG files for Junos OS conform to the YANG language syntax defined in RFC 6020 YANG 1.0 *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)* and RFC 7950 *The YANG 1.1 Data Modeling Language*. Certain directives need to be present in the file that configure XML proxy.

To use the `xmlproxyd` process to translate telemetry data, create a `render.yang` file. In this file, the `dr:command-app` is set to `xmlproxyd`.

The XML proxy YANG filename and module name must start with `xmlproxyd_`:

- For the XML proxy YANG filename, add the extension `.yang`, for example, `xmlproxyd_sysusers.yang`
- For the module name, use the filename without the extension `.yang`, for example, `xmlproxyd_sysusers`

To simplify creating a YANG file, it's easiest to start by modifying a working example.

1. Provide a name for the module. The module name must start with `xmlproxyd_` and be the same name as the XML proxy YANG file name.

   For example, for an XML proxy YANG file called `sysusers.yang`, drop the `.yang` extension and name the module `xmlproxyd_sysusers`:

   ```
   module xmlproxyd_sysusers {
   ```

2. For the Junos telemetry interface include the process name `xmlproxyd`:

   ```
   dr:command-app "xmlproxyd";
   ```

3. Include the following RPC for the NETCONF get request:

   `rpc juniper-netconf-get {`

4. Specify the location of the output of the RPC, where *company-name* is the name you give to the location:

   `dr:command-top-of-output "/company-name";`

5. Include the following command to execute the RPC:

   `dr:command-full-name "drend juniper-netconf-get";`

6. Specify the CLI command from which to retrieve data. The Junos OS CLI command that gets executed at the requested sample frequency is defined under `dr:cli-command` and executed by the `xmlproxyd` process.

   To retrieve command output for the Junos OS command `show system users`:

   `dr:cli-command "show system users";`

7. Escalate privileges, logon as “root”, connect to the internal management socket via Telnet, and specify help for an RPC:

   `dr: command-help “default <get> rpc”;`

   When this is included in the YANG file, output that is helpful for debugging is displayed in the `help drend` output on the internal management socket:

   ```
   telnet /var/run/xmlproxyd_mgmt
   Trying /var/run/xmlproxyd_mgmt...
   Connected to /var/run/xmlproxyd_mgmt.
   Escape character is '^]'.
   220 XMLPROXYD release 18.2I20180412_0904_bijchand built by bijchand on 2018-04-12 14:48:48 UTC
   help drend
   ```

   `200-juniper-netconf-get-0 system users <get> RPC`

8. Specify the hierarchy and use the `dr:source` command to map to a container, a list, or a specific leaf. The absolute path under which the sensors will be reported is built from the output group `junos` plus `system-users-information`, concatenated by `/’`. The path `/junos/system-users-information/` is the path to query for information about this custom sensor.

   WARNING: You should not create a custom YANG model that conflicts or overlaps with predefined native paths (Juniper defined paths) and OpenConfig paths (resources). Doing so can result in undefined behavior.

   For example, do not create a model that defines new leafs at or augments nodes for resource paths such as `/junos/system/linecard/firewall`or `/interfaces`.

A one-to-one mapping between container, leafs and the XML tag or value from the CLI command output is defined in the grouping referenced by `uses` within the output container. A *grouping* can be referred to multiple times in different container outputs. The container `system-users-information` below uses the grouping system-users-information. However, it is defined without the aforementioned one-to-one mapping for every container, list and leaf to an output XML tag from the CLI command XML output.

```
output {
    container junos {
        container system-users-information {
            dr:source "/system-users-information“;
            uses system-users-information-grouping;
        }
    }
}
```

9. The following YANG file shows how to include these commands to enable the `xmlproxyd` process to retrieve the full operational state and map it to the leafs in Juniper’s own data model:

```
*/



/*
 * Example yang for generating OpenConfig equivalent of show system users
 */

module xmlproxyd_sysusers {
  yang-version 1;

  namespace "http://juniper.net/yang/software";

  import drend {
    prefix dr;
  }

  grouping system-users-information-grouping {
    container uptime-information {
      dr:source "uptime-information";
      leaf date-time {
        type string;
        dr:source date-time;
      }
      leaf up-time {
        type string;
        dr:source up-time;
      }
      leaf active-user-count {
        type int32;
        dr:source active-user-count;
      }
      leaf load-average-1 {
        type string;
        dr:source load-average-1;
      }
      leaf load-average-5 {
        type string;
        dr:source load-average-5;
      }
      leaf load-average-15 {
        type string;
        dr:source load-average-15;
      }
      container user-table {
        dr:source "user-table";
        list user-entry {
          key "user";
          dr:source "user-entry";
          leaf user {
            dr:source user;
            type string;
          }
          leaf tty {
            dr:source tty;
            type string;
          }
          leaf from {
            dr:source from;
            type string;
          }
          leaf login-time {
            dr:source login-time;
            type string;
          }
          leaf idle-time {
```

```
            dr:source idle-time;
            type string;
          }
          leaf command {
            dr:source command;
            type string;
          }
        }
      }
    }
  }

  dr:command-app "xmlproxyd";
  rpc juniper-netconf-get {
    dr:command-top-of-output "/company-name";
    dr:command-full-name "drend juniper-netconf-get";
    dr:cli-command "show system users";
    dr:command-help "default <get> rpc”;
output {
      container company-name {
        container system-users-information {
          dr:source "/system-users-information";
          uses system-users-information-grouping;
        }
      }
    }
  }
}
```

Load the Yang File in Junos

After the YANG file is complete, upload the YANG file and verify that the module is created.

1. Upload the YANG file to the router.
2. Register the YANG file using the `request system yang add package` command.

```
user@switch> request system yang add package sysusers proxy-xml module xmlproxyd_sysusers.yang
XML proxy YANG module validation for xmlproxyd_sysusers.yang : START
XML proxy YANG module validation for xmlproxyd_sysusers.yang : SUCCESS
JSON generation for xmlproxyd_sysusers.yang : START
JSON generation for xmlproxyd_sysusers.yang: SUCCESS
```

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the `run` command is not supported.

3. Verify that the module (sensor) is registered using the `show system yang package sysusers` command, where `sysusers` is the name of the package:

```
user@switch> show system yang package sysusers
Package ID                :sysusers
XML Proxy YANG Module(s) :xmlproxyd_sysusers.yang
```

4. Enable gRPC in the Junos OS configuration:

```
user@switch> set system services extension-service request-response grpc port 32767
```

Collect Sensor Data

IN THIS SECTION

Use your favorite collector to pull the newly created telemetry sensor data from the device.

Consider resource constraints before initiating sensors:

- Avoid specifying the same reporting interval for multiple XML proxy sensors.
- Because xmlproxyd performs XML and text processing, a device should only contain XML proxy sensors that execute within the CPU utilization range.

Platform-Specific Collecting Sensor Data Behavior

Use Feature Explorer to confirm platform and release support for specific features.

Use the following table to review platform-specific behaviors for your platforms:

Table 34: Collecting Sensor Data Behavior

Platform	Difference
PTX10008	For PTX10008 routers operating Junos OS Evolved, do not connect more than 10 collectors per router for telemetry RPCs.

The following instructions use the collector *jtimon*. For information about jtimon setup, see Junos Telemetry Interface client.

NOTE: If a subscription already exists for a sensor and a duplicate subscription is configured, the connection between the collector and the device will close with the error message `AlreadyExists`.

1. Create a simple configuration file, here named `vmx1.json`. Adjust the host IP address and the port, as needed. The path `/junos/system-users-information` is specified. The `freq` field is defined in MicroSoft, streaming a new set of key value pairs every 5 seconds. Optionally, you can add multiple paths.

```
$ cat vmx1.json
{
  "host": "172.16.122.182
  "port": 32767
  "cid": "my-client-id",
  "grpc" : {
    "ws" : 524289
  },
  "paths": {
     {
       "path": "/junos/system-users-information/",
       "freq": 5000
     },
     {
       "path": "/junos/additional-path/",  <-OPTIONAL
       "freq": 5000
     }
   }
}
```

2. Launch the collector, using either your own compiled file or an automatically built image from Docker Hub. The sample query output below shows the sensor report by path. Every key is sent in human-readable form as an absolute path. In case of lists, the absolute path contains an index in the form of XPATH which is ideal to group values from a (time series) database, such as InfluxDB. For example, the output below shows the path `/junos/system-users-information/uptime-information/user-table/user-entry[user='ab']/`.

   You can terminate the stream of sensor data using Ctrl-C.

   ```
   $ docker run -tu --rm -v $(PWD):/u mw/jtimon --config vmx1.json --print
   gRPC headers from Junos:
     init-response: [response { subscription_id 1} path_list {path: "junos/system-users-
   information/" sample-frequency: 5000 } ]
     content-type: [application/grpc]
     grpc-accept-encoding: [identity,deflate,gzip]
   2018/03/04 17:13:19 system-id vmxdockerlight_vmx1_1
   2018/03/04 17:13:19 component_id 65535
   2018/03/04 17:13:19 sub_component_id: 0
   2018/03/04 17:13:19 path: sensor_1000:/junos/system-users-information/:/junos/system-users-
   information/
   2018/03/04 17:13:19 sequence_number: 16689
   2018/03/04 17:13:19 timestamp: 1520183589391
   2018/03/04 17:13:19 sync_response: %!d(bool=false)
   2018/03/04 17:13:19 key: __timestamp__
   2018/03/04 17:13:19 uint_value: 1520183589391
   2018/03/04 17:13:19 key: __junos_re_stream_creation_timestamp--
   2018/03/04 17:13:19 uint value: 1520183589372
   2018/03/04 17:13:19 key: __junos_re_payload-get_timestamp__
   2018/03/04 17:13:19 uint_value: 1520183589390
   2018/03/04 17:13:19 key: /junos/system-users-information/uptime-information/date-time
   2018/03/04 17:13:19 str-value: 5:13PM
   2018/03/04 17:13:19 key: /junos/system-users-inforamtion/uptime-information/up-time
   2018/03/04 17:13:19 str-value: 1 day, 4:10
   2018/03/04 17:13:19 key: /junos/system-users-information/uptime-information/active-user-count
   2018/03/04 17:13:19 int_value: 2
   2018/03/04 17:13:19 key: /junos/system-users-inforamtion/uptime-information/load-average-1
   2018/03/04 17:13:19 str_value: 0.62
   2018/03/04 17:13:19 key: /junos/system-users-information/uptime-information/load-average-5
   2018/03/04 17:13:19 str_value: 0.56
   2018/03/04 17:13:19 key: /junos/system-users-inforamtion/uptime-information/load-average-15
   2018/03/04 17:13:19 str_value: 0.53
   2018/03/04 17:13:19 key: __prefix__
   2018/03/04 17:13:19 str_value: /junos/system-users-information/uptime-information/user-table/
   ```

```
user-entry[user='ab']/
2018/03/04 17:13:19 key: tty
2018/03/04 17:13:19 str_value: pts/1
2018/03/04 17:13:19 key: from
2018/03/04 17:13:19 str-value: 172,16.04.25
2018/03/04 17:13:19 key: login-time
2018/03/04 17:13:19 str_value: 5:12PM
2018/03/04 17:13:19 key: idle-time
2018/03/04 17:13:19 str-value: -
2018/03/04 17:13:19 key: command
2018/03/04 17:13:19 str_value: -cl
2018/03/04 17:13:19 system_id: vmxdockerlight_vmx1_1
2018/03/04 17:13:19 component_id: 65535
2018/03/04 17:13:19 sub_component_id: 0
2018/03/04 17:13:19 <output truncated>
```

The sample query shown below shows two sensor reports per path, then I terminated it with Ctrl-C. Every key is sent in human readable form as an absolute path and in case of lists, contains an index in form of XPATH, ideal to group values from a (time series) database like InfluxDB e.g. /junos/system-users-information/uptime-information/user-table/user-entry[user='ab']/

3. Verify that the module (sensor) is loaded using the `show system yang package sysusers` command, where `sysusers` is the name of the package:

```
user@switch> show system yang package sysusers
Package ID                :sysusers
XML Proxy YANG Module(s) :xmlproxyd_sysusers.yang
```

4. Enable gRPC in the Junos OS configuration:

```
user@switch> set system services extension-service request-response grpc port 32767
```

Installing a User-Defined YANG File

To add, validate, modify, or delete a user-defined YANG file for XML proxy for the Junos telemetry interface, use the `request system yang` set of commands from the operational mode:

1. Specify the name of the XML proxy YANG file and the file path to install it. This command creates a `.json` file in the `/opt/lib/render` directory.

```
user@switch> request system yang add package package-name proxy-xml module file-path-name
```

NOTE: This command can be performed only on the current routing engine.

To add multiple YANG modules with the `request system yang add package package-name proxy-xml module` command, enclose the *file-path-name* in brackets: `[ file-path-name 1 file-path-name 2 ]`

2. (Optional) Validate an module before adding it to the router using the **request system yang validate proxy-xml module *module-name*** command. .

```
user@switch> request system yang validate proxy-xml module module-name
```

The output `XML proxy YANG module validation for xmlproxyd_<module-name> : SUCCESS` indicates successful module validation.

Mismatch error sometimes occur. If the command returns the error below, you can eliminate the error by using Junos OS Release 18.1R1 or later:

```
user@switch> request system yang validate proxy-xml module xmlproxyd_sysusers.yang
error: illegal identifier <identifier> , must not start with [xX][mM][lL]
```

3. (Optional) Update an existing XML proxy YANG file that was previously added.

```
user@switch> request system yang update package-name proxy-xml module file-path-name
```

4. Delete an existing XML proxy YANG file.

```
user@switch> request system yang delete package-name
```

5. Verify that the YANG file has been installed by entering the `show system yang package` command.

```
user@switch> show system yang package package-name
```

Troubleshoot Telemetry Sensors

IN THIS SECTION

Problem

Description

Use the following methods to troubleshoot user-define telemetry sensors:

- Execute a tcpdump for the interface your gRPC requests came from (for this task, interface `fxp0` was used).

```
user@switch>monitor traffic interface fxp0 no-resolve matching "tcp port 32767"
```

- Enable traceoptions using the **set services analytics traceoptions flag xmlproxy** command. Check the `xmlproxyd` log file for confirmation of whether the CLI command's RPC was sent and if a response was received:

1. Issue the **show log xmlproxyd** command to show the xmlproxyd log. The value for the field `xmlproxy_execute_cli_command:` indicates if the RPC was sent or not. The value for the field `xmlproxy_build_context` indicates the command.

```
user@switch>show log xmlproxyd
Mar 4 18:52:46 vmxdockerlight_vmx1_1 clear-log[52495]: logfile cleared
Mar  4 18:52:51 xmlproxy_telemetry_start_streaming: sensor /junos/system-users-information/
Mar  4 18:52:51 xmlproxy_build_context: command show system users merge-tag:
Mar  4 18:52:51 <command format="xml">show system users</command>
Mar  4 18:52:51 xmlproxy_execute_cli_command: Sent RPC..
Mar  4 18:52:51 <system-users-information xmlns="http://xml.juniper.net/junos/17.4R1/junos"
xmlns:junos="http://xml.juniper.net/junos/*/junos">
<uptime-information>
<date-time junos:seconds="1520189571">
6:52PM
</date-time>
<up-time junos:seconds="107400">
```

```
1 day,  5:50
</up-time>
<active-user-count junos:format="1 users">
1
</active-user-count>
<load-average-1>
0.94
</load-average-1>
<load-average-5>
0.73
</load-average-5>
<load-average-15>
0.65
```

RELATED DOCUMENTATION

Understanding YANG on Devices Running Junos OS | **470**

Guidelines for gRPC and gNMI Sensors (Junos Telemetry Interface)

Send Requests to the NETCONF Server | **124**

7

PART

OpenDaylight Integration

CHAPTER 19

Configure OpenDaylight Integration

IN THIS CHAPTER

Configure Interoperability Between MX Series Routers and OpenDaylight

IN THIS SECTION

OpenDaylight (ODL), hosted by the Linux Foundation, is an open-source platform for network programmability aimed at enhancing software-defined networking (SDN).

You can configure interoperability between MX Series routers and the ODL controller. ODL provides a southbound Network Configuration Protocol (NETCONF) connector API, which uses NETCONF and YANG models, to interact with a network device. A southbound interface, an OpenFlow (or alternative) protocol specification, enables communication between ODL and routers or switches. After you configure interoperability between the ODL controller and the router, you can use the ODL platform to change the router configuration, orchestrate and provision the router, and execute remote procedure calls (RPCs) on the router to get state information.

Setting up interoperability between ODL and an MX Series router involves the following tasks:

Configuring NETCONF on the MX Series Router

As a prerequisite for configuring interoperability between ODL and an MX Series router, you must configure NETCONF on the router. NETCONF is used by the ODL controller to interact with southbound devices.

To configure NETCONF on the router:

1. Enable access to the NETCONF SSH subsystem.

```
[edit]
user@host# set system services netconf ssh
```

2. Configure the NETCONF server to enforce certain behaviors that are compliant with RFC 4741, *NETCONF Configuration Protocol*, during NETCONF sessions.

```
[edit]
user@host# set system services netconf rfc-compliant
```

3. Configure the `yang-compliant` statement to require that the NETCONF server return YANG-compatible configuration data for the `<get-config>` and `<get-configuration format="xml">` RPCs.

```
[edit]
user@host# set system services netconf yang-compliant
```

4. Commit the changes.

```
[edit]
user@host# commit
```

Configuring NETCONF Trace Options

After you configure NETCONF on the router, you must configure NETCONF trace options. For more information about NETCONF and Junos XML protocol tracing operations, see "NETCONF and Junos XML Protocol Tracing Operations Overview" on page 175.

To configure NETCONF trace options:

Configure the details of the file to receive the output of the tracing operation. You can configure the file name, maximum file size, and flags to indicate tracing operations, by using the following statements:

```
[edit]
user@host# set system services netconf traceoptions file file name
user@host# set system services netconf traceoptions file size size
user@host# set system services netconf traceoptions flag flag
user@host# commit
```

To know more about configuring tracing operations for NETCONF and Junos XML protocol sessions, see this example.

Connecting ODL to MX Series Router

After NETCONF is configured on the MX Series router, you need to connect the ODL controller to the router to complete the process. For more details on this, see this ODL documentation.

RELATED DOCUMENTATION

NETCONF and Junos XML Protocol Tracing Operations Overview | **175**

NETCONF Session Overview | **35**

8

PART

Configuration Statements and Operational Commands

Junos CLI Reference Overview

We've consolidated all Junos CLI commands and configuration statements in one place. Read this guide to learn about the syntax and options that make up the statements and commands. Also understand the contexts in which you'll use these CLI elements in your network configurations and operations.

- Junos CLI Reference

Click the links to access Junos OS and Junos OS Evolved configuration statement and command summary topics.

- Configuration Statements
- Operational Commands