JUNIPER NETWORKS

JUNOS® OS

NETCONF XML Management Protocol Guide

Release
10.4

Published: 2012-03-15

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

JUNOS® OS NETCONF XML Management Protocol Guide
Release 10.4

Writing: Tony Mauro, Andrea Couvrey, Michael Scruggs, Brenda Wilden
Editing: Stella Hackell, Nancy Kurahashi, Sonia Saruba, Laura Singer
Illustration: Faith Bradford
Cover Design: Edmonds Design

Revision History
October 2010 —R1 Junos 10.4

The information in this document is current as of the date on the title page.

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

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at http://www.juniper.net/support/eula.html. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Abbreviated Table of Contents

Table of Contents

List of Tables

About This Guide

This preface provides the following guidelines for using the *JUNOS® OS NETCONF XML Management Protocol Guide*:

Junos Documentation and Release Notes

For a list of related Junos documentation, see http://www.juniper.net/techpubs/software/junos/ .

If the information in the latest release notes differs from the information in the documentation, follow the *Junos Release Notes.*

To obtain the most current version of all Juniper Networks® technical documentation, see the product documentation page on the Juniper Networks website at http://www.juniper.net/techpubs/ .

Juniper Networks supports a technical book program to publish books by Juniper Networks engineers and subject matter experts with book publishers around the world. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration using the Junos operating system (Junos OS) and Juniper Networks devices. In addition, the Juniper Networks Technical Library, published in conjunction with O'Reilly Media, explores improving network security, reliability, and availability using Junos OS configuration techniques. All the books are for sale at technical bookstores and book outlets around the world. The current list can be viewed at http://www.juniper.net/books .

Objectives

This guide describes how to use the NETCONF Extensible Markup Language (XML) management protocol and the Junos XML application programming interface (API) to

configure or request information from the NETCONF server on a device running Junos OS..

NOTE: For additional information about the Junos OS—either corrections to or information that might have been omitted from this guide—see the software release notes at http://www.juniper.net/ .

Audience

This guide is designed for network administrators who are configuring and monitoring a Juniper Networks M Series, MX Series, T Series, EX Series, or J Series router or switch.

This guide is designed for Juniper Networks customers who want to write custom applications for configuring or monitoring a Juniper Networks device that runs the Junos OS. It assumes that you are familiar with basic terminology and concepts of XML, with XML-parsing utilities such as the Document Object Model (DOM) or Simple API for XML (SAX), and with the Junos OS command-line interface (CLI).

To use this guide, you need a broad understanding of networks in general, the Internet in particular, networking principles, and network configuration. You must also be familiar with one or more of the following Internet routing protocols:

- Border Gateway Protocol (BGP)
- Distance Vector Multicast Routing Protocol (DVMRP)
- Intermediate System-to-Intermediate System (IS-IS)
- Internet Control Message Protocol (ICMP) router discovery
- Internet Group Management Protocol (IGMP)
- Multiprotocol Label Switching (MPLS)
- Open Shortest Path First (OSPF)
- Protocol-Independent Multicast (PIM)
- Resource Reservation Protocol (RSVP)
- Routing Information Protocol (RIP)
- Simple Network Management Protocol (SNMP)

Personnel operating the equipment must be trained and competent; must not conduct themselves in a careless, willfully negligent, or hostile manner; and must abide by the instructions provided by the documentation.

Supported Platforms

For the features described in this manual, Junos OS currently supports the following platforms:

- EX Series
- J Series
- M Series
- MX Series
- SRX Series
- T Series

Using the Indexes

This reference contains two indexes: a standard index with topic entries, and an index of commands.

Documentation Conventions

Table 1 on page xv defines notice icons used in this guide.

Table 1: Notice Icons

Icon	Meaning	Description
	Informational note	Indicates important features or instructions.
	Caution	Indicates a situation that might result in loss of data or hardware damage.
	Warning	Alerts you to the risk of personal injury or death.
	Laser warning	Alerts you to the risk of personal injury from a laser.

Table 2 on page xvi defines the text and syntax conventions used in this guide.

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the **configure** command: user@host> **configure**
`Fixed-width text like this`	Represents output that appears on the terminal screen.	`user@host>` **show chassis alarms** `No alarms currently active`
Italic text like this	• Introduces important new terms. • Identifies book names. • Identifies RFC and Internet draft titles.	• A policy *term* is a named structure that defines match conditions and actions. • *Junos OS System Basics Configuration Guide* • RFC 1997, *BGP Communities Attribute*
Italic text like this	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# **set system domain-name** *domain-name*
Text like this	Represents names of configuration statements, commands, files, and directories; interface names; configuration hierarchy levels; or labels on routing platform components.	• To configure a stub area, include the **stub** statement at the **[edit protocols ospf area area-id]** hierarchy level. • The console port is labeled **CONSOLE**.
< > (angle brackets)	Enclose optional keywords or variables.	**stub** <**default-metric** *metric*>**;**
| (pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	**broadcast | multicast** (***string1*** **|** ***string2*** **|** ***string3***)
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	**rsvp { # Required for dynamic MPLS only**
[ ] (square brackets)	Enclose a variable for which you can substitute one or more values.	**community name members [** ***community-ids*** **]**

Indention and braces ({ })	Identify a level in the configuration hierarchy.	[edit]
		routing-options {
		  static {
		  route default {
		  nexthop *address*;
		  retain;
		  }
		  }
		}

; (semicolon)	Identifies a leaf statement at a configuration hierarchy level.	

J-Web GUI Conventions

Table 2: Text and Syntax Conventions *(continued)*

Convention	Description	Examples
Bold text like this	Represents J-Web graphical user interface (GUI) items you click or select.	• In the Logical Interfaces box, select **All Interfaces**. • To cancel the configuration, click **Cancel**.
> (bold right angle bracket)	Separates levels in a hierarchy of J-Web selections.	In the configuration editor hierarchy, select **Protocols>Ospf**.

Documentation Feedback

We encourage you to provide feedback, comments, and suggestions so that we can improve the documentation. You can send your comments to techpubs-comments@juniper.net, or fill out the documentation feedback form at https://www.juniper.net/cgi-bin/docbugreport/ . If you are using e-mail, be sure to include the following information with your comments:

- Document or topic name
- URL or page number
- Software release version (if applicable)

Requesting Technical Support

Technical product support is available through the Juniper Networks Technical Assistance Center (JTAC). If you are a customer with an active J-Care or JNASC support contract, or are covered under warranty, and need post-sales technical support, you can access our tools and resources online or open a case with JTAC.

- JTAC policies—For a complete understanding of our JTAC procedures and policies, review the *JTAC User Guide* located at http://www.juniper.net/us/en/local/pdf/resource-guides/7100059-en.pdf .
- Product warranties—For product warranty information, visit http://www.juniper.net/support/warranty/ .
- JTAC hours of operation—The JTAC centers have resources available 24 hours a day, 7 days a week, 365 days a year.

Self-Help Online Tools and Resources

For quick and easy problem resolution, Juniper Networks has designed an online self-service portal called the Customer Support Center (CSC) that provides you with the following features:

- Find CSC offerings: http://www.juniper.net/customers/support/
- Search for known bugs: http://www2.juniper.net/kb/

- Find product documentation: http://www.juniper.net/techpubs/
- Find solutions and answer questions using our Knowledge Base: http://kb.juniper.net/
- Download the latest versions of software and review release notes: http://www.juniper.net/customers/csc/software/
- Search technical bulletins for relevant hardware and software notifications: https://www.juniper.net/alerts/
- Join and participate in the Juniper Networks Community Forum: http://www.juniper.net/company/communities/
- Open a case online in the CSC Case Management tool: http://www.juniper.net/cm/

To verify service entitlement by product serial number, use our Serial Number Entitlement (SNE) Tool: https://tools.juniper.net/SerialNumberEntitlementSearch/

Opening a Case with JTAC

You can open a case with JTAC on the Web or by telephone.

- Use the Case Management tool in the CSC at http://www.juniper.net/cm/ .
- Call 1-888-314-JTAC (1-888-314-5822 toll-free in the USA, Canada, and Mexico).

For international or direct-dial options in countries without toll-free numbers, see http://www.juniper.net/support/requesting-support.html .

PART 1

Overview

CHAPTER 1

Introduction to the NETCONF XML Management Protocol and Junos XML API

This chapter discusses the following:

XML and the Junos OS

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, the 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 configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos 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.

You can view the XML-formatted output of any operational mode command by issuing the command in the CLI and adding the **| display xml** option. The following example shows the text-fomatted and XML-formatted 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/10.4B1/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/10.4B1/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

You can view the Junos XML API representation of any operational mode command by issuing the command in the CLI and adding the **| display xml rpc** option. The following example shows the Junos XML API tag element for the **show chassis alarms** command.

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4B1/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 command's corresponding Junos XML API request tag element that is sent to the 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 its equivalent Junos XML API request tag **<get-alarm-information>** and sends the XML request to the Junos infrastructure for processing. The 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 automation scripts use XML to communicate with the host device. The Junos OS provides XML-formatted input to a script. The script processes the input and then returns XML-formatted output to the 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 the Junos OS for processing. Op script input consists of a blank XML document. Event scripts receive an XML document containing the description of the triggering event. Commit script input consists of an XML representation of the postinheritance candidate configuration file.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview
- XML Overview
- *Junos XML API Configuration Reference*

- *Junos XML API Operational Reference*

NETCONF XML Management Protocol and Junos XML API Overview

The NETCONF XML management protocol is an XML-based management protocol that client applications use to request and change configuration information on routing, switching, and security platforms running Junos OS. The NETCONF XML management protocol uses an Extensible Markup Language (XML) based data encoding for the configuration data and remote procedure calls. The NETCONF XML management protocol defines basic operations that are equivalent to configuration mode commands in the Junos OS command-line interface (CLI). Applications use the protocol operations to display, edit, and commit configuration statements (among other operations), just as administrators use CLI configuration mode commands such as **show**, **set**, and **commit** to perform those operations.

The Junos XML API is an XML representation of Junos configuration statements and operational mode commands. Junos XML configuration tag elements are the content to which the NETCONF XML protocol operations apply. Junos XML operational tag elements are equivalent in function to operational mode commands in the CLI, which administrators use to retrieve status information for a routing platform.

The NETCONF XML management protocol is described in RFC 4741, *NETCONF Configuration Protocol*, available at http://www.ietf.org/rfc/rfc4741.txt .

Client applications request or change information on a switch, router, or security device by encoding the request with tag elements from the NETCONF XML management protocol and Junos XML API and sending it to the NETCONF server on the device. The NETCONF server is integrated into the Junos OS and does not appear as a separate entry in process listings. The NETCONF server directs the request to the appropriate software modules within the device, encodes the response in NETCONF and Junos XML API tag elements, and returns the result to the client application. For example, to request information about the status of a device's interfaces, a client application sends the **<get-interface-information>** tag element from the Junos XML API. The NETCONF server gathers the information from the interface process and returns it in the **<interface-information>** tag element.

You can use the NETCONF XML management protocol and Junos XML API to configure devices running Junos OS or request information about the device configuration or operation. You can write client applications to interact with the NETCONF server, but you can also use the NETCONF XML management protocol to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

XML Overview

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. Tags look much like Hypertext Markup Language (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 Junos XML and NETCONF XML management protocol tag elements:

Junos XML and NETCONF XML Management Protocol Tag Elements

Items in an XML-compliant document or data set are always enclosed in paired opening and closing tags. 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:

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

The term *tag 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 a tag 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 a Junos XML tag element or NETCONF tag element in its opening and closing tags. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in Juniper Networks documentation to indicate optional parts of CLI command strings.

NETCONF and Junos XML tag elements obey the XML convention that the tag element name indicates the kind of information enclosed by the tag element. For example, the name of the Junos XML **<interface-state>** tag element indicates that it contains a description of the current status of an interface on the routing platform, whereas the

name of the **<input-bytes>** tag element indicates that its contents specify the number of bytes received.

When discussing tag elements in text, this documentation conventionally uses just the name of the opening tag to represent the complete tag 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>** tag element.

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. The rules define the contexts in which each tagged item can—and in some cases must—occur. A file called a *document type definition*, or *DTD*, lists every tag element that can appear in the document or data set, defines the parent-child relationships between the tags, and specifies other tag characteristics. The same DTD can apply to many XML documents or data sets.

Advantages of Using the NETCONF XML Managment Protocol and Junos XML API

The NETCONF XML management protocol and Junos XML API fully document all options for every supported Junos operational request and all elements in every Junos 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 or document. NETCONF and Junos XML tag elements make it straightforward for client applications that request information from a device to parse the output and find specific information.

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

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, SNMP ifIndex: 3
```

The corresponding XML-tagged version is:

```
<interface>
  <name>fxp0</name>
  <admin-status>enabled</admin-status>
  <operational-status>up</operational-status>
  <index>4</index>
  <snmp-index>3</snmp-index>
</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 one difficulty is that 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, but must instead 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, if a **Logical index** field is added following the interface index number:

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, Logical index: 12, SNMP ifIndex: 3
```

An application that extracts the interface index number delimited by the **Interface index:** and **SNMP ifIndex** labels now obtains an incorrect result. The application must be updated manually to 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 **<index>** tag and closing **</index>** tag. The application does not have to rely on an element's position in the output string, so the NETCONF server can emit the child tag elements in any order within the **<interface>** tag element. Adding a new **<logical-index>** tag element in a future release does not affect an application's ability to locate the **<index>** tag element and extract its contents.

Tagged output is also easier to transform into different display formats. For instance, 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 have to design and write special routines and data structures in your display program to extract and store 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. It is also easy to use the same extraction routine for several levels of detail, simply ignoring the tag elements you do not need when creating a less detailed display.

Overview of a NETCONF XML Management Protocol Session

Communication between the NETCONF server and a client application is session based. The two parties explicitly establish a connection before exchanging data and close the connection when they are finished. Each request from the client application and each response from the NETCONF server constitutes a *well-formed* XML document, because the tag streams obey the structural rules defined in the NETCONF and Junos XML DTDs for the kind of information they encode. Client applications must produce a well-formed XML document for each request by emitting tag elements in the required order and only in the legal contexts.

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.

CHAPTER 2

Using NETCONF XML Management Protocol and Junos XML Tag Elements

This chapter describes the syntactic and notational conventions used by the NETCONF server and client applications, including the mappings between statements and commands in the Junos OS command-line interface (CLI) and the tag elements in the Junos Extensible Markup Language (XML) application programming interface (API).

For more information about the syntax of CLI commands and configuration statements, see the *Junos OS CLI User Guide*. For information about specific configuration statements, see the Junos OS configuration guides. For information about specific operational mode commands, see the Junos OS command references.

This chapter discusses the following topics:

XML and NETCONF XML Management Protocol Conventions Overview

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 DTDs 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 XML management 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 element. For information about the **<rpc>** tag element, see "Sending a Request to the NETCONF Server" on page 43.

A *response* tag element represents the NETCONF server's reply to a request tag element and occurs only within an **<rpc-reply>** tag element. For information about the **<rpc-reply>** tag element, see "Parsing the NETCONF Server Response" on page 46.

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>
]]>]]>
```

T2100

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 13.

For information about the attributes in the opening **<rpc-reply>** tag, see "Parsing the NETCONF Server Response" on page 46. For information about the **xmlns** attribute in the opening **<interface-information>** tag, see "Requesting Operational Information" on page 60. For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 28.

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 16.

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 3 on page 14 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 3: Predefined Entity Reference Substitutions for Tag Content Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	**&**
> (greater-than sign)	**>**
< (less-than sign)	**<**

Table 4 on page 15 summarizes the mapping between disallowed characters and predefined entity references for attribute values.

Table 4: Predefined Entity Reference Substitutions for Attribute Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	**&**
' (apostrophe)	**'**
> (greater-than sign)	**>**
< (less-than sign)	**<**
" (quotation mark)	**"**

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">
```

Mapping Commands to Junos XML Tag Elements

The Junos XML API defines tag-element equivalents for many commands in CLI operational mode. For example, the **<get-interface-information>** tag element corresponds to the **show interfaces** command.

Information about the available command equivalents in the current release of the Junos OS can be found in the *Junos XML API Operational Reference*. For the mapping between commands and Junos XML tag elements, see the *Junos XML API Operational Reference* "Mapping Between Operational Tag Elements, Perl Methods, and CLI Commands" chapter. For detailed information about a specific operation, see the *Junos XML API Operational Reference* "Summary of Operational Request Tags" chapter.

The following sections describe the tag elements that map to command options:

- Mapping for Command Options with Variable Values on page 16
- Mapping for Fixed-Form Command Options on page 16

Mapping for Command Options with Variable Values

Many CLI commands have options that identify the object that the command affects or reports about, 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 learn 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 or in the *Junos XML API Operational Reference*.

The following example shows the Junos XML tag elements for two CLI operational commands that have variable-form options. In the **show interfaces** command, **t3-5/1/0:0** 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.

CLI Command	JUNOS XML Tags
show interfaces t3-5/1/0:0	<rpc> <get-interface-information> <interface-name>t3-5/1/0:0</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>

T1500

Mapping for Fixed-Form Command Options

Some CLI commands include options that have a fixed form, such as the **brief** and **detail** strings, 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	**JUNOS XML Tags**
show isis adjacency detail	<rpc> <get-isis-adjacency-information> <detail/> </get-isis-adjacency-information> </rpc>

T1501

Mapping Configuration Statements to Junos XML Tag Elements

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>** tag element is the top-level Junos XML container tag 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>
```

T1502

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 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 19, "Mapping for Leaf Statements with Multiple Values" on page 20, and "Mapping for Multiple Options on One or More Lines" on page 20.

The following example shows the Junos XML tag elements for configuration statements that define two BGP groups called **G1** and **G2**. Notice that the Junos XML **<name>** tag 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. For complete information about changing routing platform configuration, see "Changing Configuration Information" on page 85.

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>
```

T1503

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;
      ...other 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 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. For complete information about changing routing platform configuration, see "Changing Configuration Information" on page 85.

CLI Configuration Statements

```
protocols {
   bgp {
      group 23 {

         import [ policy1 policy2 ];

      }
   }
}
```

JUNOS XML Tags

```
<configuration>
   <protocols>
      <bgp>
         <group>
            <name>23</name>
            <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

```
system {
   login {
      class user-accounts {

         permissions [ configure admin control ];


      }
   }
}
```

JUNOS XML Tags

```
<configuration>
   <system>
      <login>
         <class>
            <name>user-accounts</name>
            <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 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 configuration can include comments that describe statements in the configuration. In CLI configuration mode, the **annotate** command specifies 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 *Junos OS CLI User Guide*.

The Junos XML API encloses comments about configuration statements in the **<junos:comment>** tag 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>** tag element immediately precedes the tag 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>** tag 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>** tag element, so in the configuration file the comment directly follows the associated statement on the same line.

CLI Configuration Statements

```
protocols {
   ospf {

      /* New backbone area */
      area 0.0.0.0 {

         interface so-0/0/0 { # From jnpr1 to jnpr2

            hello-interval 5;
         }
      }
   }
}
```

JUNOS XML Tags

```
<configuration>
   <protocols>
      <ospf>
         <junos:comment>
               /* New backbone area */
         </junos:comment>
         <area>
            <name>0.0.0.0</name>
            <junos:comment> # From jnpr1 to jnpr2</junos:comment>
            <interface>
               <name>so-0/0/0</name>
               <hello-interval>5</hello-interval>
            </interface>
         </area>
      </ospf>
   </protocols>
</configuration>
```

T1510

Using the Same Configuration Tag Elements in Requests and Responses

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. For more information about changing routing platform configuration, see "Changing Configuration Information" on page 85.

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 "Requesting Configuration Information" on page 61.

The tag streams emitted by the NETCONF server and by a client application can differ in the use of white space, as described in "Spaces, Newline Characters, and Other White Space" on page 13.

PART 2

Using the NETCONF XML Management Protocol

CHAPTER 3

Controlling the NETCONF Session

This chapter explains how to start and terminate a session with the NETCONF server, and describes the Extensible Markup Language (XML) tag elements from the NETCONF XML management protocol that client applications and the NETCONF server use to coordinate information exchange during the session. It discusses the following topics:

Client Application's Role in a NETCONF Session

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

1. Establishes a connection to the NETCONF server on the routing platform, as described in "Connecting to the NETCONF Server" on page 39.
2. Opens a NETCONF session, as described in "Starting the NETCONF Session" on page 40.
3. (Optional) Locks the candidate configuration, as described in "Locking the Candidate Configuration" on page 50. Locking the configuration prevents other users or applications from changing it at the same time.
4. Requests operational or configuration information, or changes configuration information, as described in "Requesting Information" on page 59 and "Changing Configuration Information" on page 85.
5. (Optional) Verifies the syntactic correctness of a configuration before attempting to commit it, as described in "Verifying a Configuration Before Committing It" on page 109.

6. Commits changes made to the configuration, as described in "Committing a Configuration" on page 110.
7. Unlocks the candidate configuration if it is locked, as described in "Unlocking the Candidate Configuration" on page 51.
8. Ends the NETCONF session and closes the connection to the device, as described in "Ending a NETCONF Session and Closing the Connection" on page 53.

Establishing a NETCONF Session

The NETCONF server communicates with client applications within the context of a NETCONF *session*. 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 access the NETCONF server using the SSH protocol and use the standard SSH authentication mechanism. After authentication, the NETCONF server uses the Junos login usernames and classes already configured on the device to determine whether a client application is authorized to make each request.

For information about establishing a connection and NETCONF session, see the following sections:

- Generating Well-Formed XML Documents on page 28
- Prerequisites for Establishing an SSH Connection on page 29
- Connecting to the NETCONF Server on page 39
- Starting the NETCONF Session on page 40

For an example of a complete NETCONF session, see "Example of a NETCONF Session" on page 54.

Generating 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 by obeying 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.

The NETCONF server and client applications must also comply with RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, available at http://www.ietf.org/rfc/rfc4742.txt . In particular, the server and applications must send the character sequence **]]>]]>** after each XML document. Because this sequence is not legal within an XML document, it unambiguously signals the end of a 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.

NOTE: In the following example (and in all examples in this document of tag elements emitted by a client application), bold font is used to highlight the part of the tag sequence that is discussed in the text.

```
<!-- generated by a 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>
]]>]]>
```

Prerequisites for Establishing an SSH Connection

You use the SSH protocol to establish connections between a *configuration management server* and a device running Junos OS. A configuration management server, as the name implies, is used to configure the device running Junos OS remotely. This server typically manages the configurations using Perl scripts.

There are two options available when establishing a connection between the configuration management server and a device running Junos OS: SSH and outbound SSH. With SSH, the configuration management server initiates an SSH session with the device running Junos OS. Outbound SSH is used when the configuration management server cannot initiate an SSH connection because of network restrictions (such as a firewall). In this situation, the device running Junos OS is configured to initiate, establish, and maintain an SSH connection with a predefined set of configuration management servers. For a complete discussion of outbound SSH, see “Configuring Outbound SSH Service” in the *Junos OS System Basics Configuration Guide*.

- Establishing an SSH Connection on page 29
- Establishing an Outbound SSH Connection on page 34

Establishing an SSH Connection

Before the configuration management server establishes an SSH connection with a device running Junos OS, you must satisfy the requirements discussed in the following sections.

- SSH Software Is Installed on the Configuration Management Server on page 30
- Client Application Can Log In on Devices Running Junos OS on page 30
- Junos Login Account Has Public/Private Key Pair or Password on page 31

- Client Application Can Access the Keys or Password on page 32
- NETCONF Service over SSH Is Enabled on page 33

SSH Software Is Installed on the Configuration Management Server

The configuration management server handles the SSH connection between the configuration management server and the device running Junos OS. Therefore, the SSH software must be installed locally on the configuration management server. If the client application accessing the NETCONF server uses the NETCONF Perl module provided by Juniper Networks, no further action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSH software. If the client application does not use the NETCONF Perl module, obtain the SSH software and install it on the configuration management server where the client application runs. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/ .

Client Application Can Log In on Devices Running Junos OS

When establishing a NETCONF session, the configuration management server must log in to the device running the Junos OS. Thus, each configuration management server needs a user account on each device where a NETCONF session will be established. The following instructions explain how to create a Junos OS login account for the configuration management server. Alternatively, you can skip this section and enable authentication through RADIUS or TACACS+; for instructions, see the chapter about system authentication in the *Junos OS System Basics Configuration Guide*.

To determine whether a login account exists on a device running Junos OS, enter CLI configuration mode on the device and issue the following commands:

```
[edit]
user@host# edit system login
[edit system login]
user@host# show user account-name
```

If the appropriate account does not exist, perform the following steps to create one:

1. Include the **user** statement at the **[edit system login]** hierarchy level and specify a username. Also include the **class** statement at the **[edit system login user *username*]** hierarchy level, and specify a login class that has the permissions required for all actions to be performed by the application. Optionally, include the **full-name** and **uid** statements.

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

   

   NOTE: For detailed information about creating user accounts, see the chapter about configuring user access in the *Junos OS System Basics Configuration Guide*.

2. Commit the configuration to activate the user account on the device. If you are going to make more changes to the configuration file, you can wait to commit the file until

you have completed making all of the changes to the file. However, the user account is not activated on the device until you commit the configuration.

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

3. Repeat the preceding steps on each device where the client application will establish a NETCONF session.

Junos Login Account Has Public/Private Key Pair or Password

The configuration management server needs an SSH public/private key pair, a text-based password, or both before it can authenticate with the NETCONF server. A public/private key pair 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+, as described in the chapter about system authentication in the *Junos OS System Basics Configuration Guide*.

Follow the instructions in the appropriate section:

- Creating a Text-Based Password on page 31
- Creating a Public/Private Key Pair on page 32

Creating a Text-Based Password

To create a text-based password, perform the following steps:

1. Include either the **plain-text-password** or **encrypted-password** statement at the **[edit system login user *account-name* authentication]** hierarchy level. First, move to that hierarchy level:

   ```
   [edit system login]
   user@host# edit user account-name authentication
   ```

   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 account-name authentication]
   user@host# set plain-text-password
   New password: password
   Retype new password: password
   ```

   To store a password that you have previously created and hashed using Message Digest 5 (MD5) or Secure Hash Algorithm 1 (SHA-1), issue the following command:

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

2. (Optional) Commit the configuration. Alternatively, you can wait until you have added the statements that satisfy all prerequisites (see "NETCONF Service over SSH Is Enabled" on page 33).

[edit system login user *account-name* authentication]
user@host# **commit**

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

Creating a Public/Private Key Pair

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

1. Issue the **ssh-keygen** command in the standard command shell (not the Junos OS CLI) on the configuration management server where the client application runs. By providing the appropriate arguments, you encode the public key with either RSA (supported by SSH versions 1 and 2) or the Digital Signature Algorithm (DSA, supported by SSH version 2). For more information, see the manual page for the **ssh-keygen** command. The Junos OS uses SSH version 2 by default, but also supports version 1.

   % **ssh-keygen** ***options***

2. Associate the public key with the Junos login account by including the **load-key-file** statement at the **[edit system login user *account-name* authentication]** hierarchy level. The Junos OS copies the contents of the specified file onto the device running Junos OS.

   [edit system login user *account-name* authentication]
   user@host# **set load-key-file** ***URL***

   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 (DSA or RSA) and SSH version. For information about specifying URLs, see the *Junos OS CLI User Guide*.

   Alternatively, you can include one or both of the **ssh-dsa** and **ssh-rsa** statements 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. For more information about the **ssh-dsa** and **ssh-rsa** statements, see the *Junos OS System Basics Configuration Guide*.

3. (Optional) Commit the configuration. Alternatively, you can wait until you have added the statements that satisfy all prerequisites (see "NETCONF Service over SSH Is Enabled" on page 33).

   [edit system login user *account-name* authentication]
   user@host# **commit**

4. Repeat Step 2 and Step 3 on each device running Junos OS where the client application establishes NETCONF sessions.

Client Application Can Access the Keys or Password

The client application must be able to access the public/private keys or password you created in "Junos Login Account Has Public/Private Key Pair or Password" on page 31 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 computer 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.

NETCONF Service over SSH Is Enabled

RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, requires that the NETCONF server, by default, provide the client device with SSH access to the server over a dedicated TCP port. Use of a dedicated port makes it easy to identify and filter NETCONF traffic. The Internet Assigned Numbers Authority (IANA) port for NETCONF-over-SSH sessions is 830.

You also can enable access to the NETCONF server either over the default SSH port (22) or over a port number that is explicitly configured on the device running Junos OS. If you specify the NETCONF-over-SSH port number in the device configuration, the configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests. The default SSH port (22) continues to accept NETCONF sessions even when an alternate NETCONF-over-SSH port is configured. For added security, you can configure event policies that utilize **UI_LOGIN_EVENT** information to effectively disable the default port or further restrict NETCONF server access on a port.

To enable NETCONF service over SSH, perform the following steps:

1. Include one of the following statements at the indicated hierarchy level:
   - To enable SSH access over the default NETCONF-over-SSH port (830) as specified by RFC 4742, include the **ssh** statement at the **[edit system services netconf]** hierarchy level:

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

   - To enable SSH access over a specified port number, configure the **port** statement with the desired 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 NETCONF-over-SSH can be configured on any port from 1 through 65535, you should avoid configuring access on a port that is normally assigned for another service. This practice avoids potential resource conflicts. If you configure NETCONF-over-SSH on 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 to that effect.

- To enable access over the default SSH port (22), include the **ssh** statement at the **[edit system services]** hierarchy level. This configuration enables SSH access to the device for all users and applications. The **ssh** statement can be included in the configuration in addition to the NETCONF-over-SSH configuration statements listed above.

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

2. Commit the configuration:

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

3. Repeat the preceding steps on each device running Junos OS where the client application will establish NETCONF sessions.

Establishing an Outbound SSH Connection

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

- Configuring the Device Running Junos OS for Outbound SSH on page 34
- Installing SSH Software on the Client on page 37
- Receiving and Managing the Outbound SSH Initiation Sequence on the Client on page 37
- Enabling NETCONF Service over SSH on page 38

Configuring the Device Running Junos OS for Outbound SSH

To configure the device running Junos OS 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 key pair for the device running Junos OS. This key pair will be used to encrypt the data transferred across the SSH connection. For more information on generating key pairs, see the *Junos OS System Basics Configuration Guide*.

3. If the public key will be installed on the configuration management server manually, transfer the public key to the configuration management server.

4. Add the following **outbound-ssh** statement at the **[edit system services]** hierarchy level:

```
[edit system services]
outbound-ssh {
  client client-id {
    device-id device-id;
    secret secret;
    keep-alive {
      retry number
      timeout number;
    }
    reconnect-strategy (sticky | in-order);
    services netconf;
    [ address ] {
      port destination-port;
      retry number;
      timeout number;
    }
  }
  traceoptions {
    file filename {
      files files;
      size size;
      match match;
      (world-readable | no-world-readable);
    }
    flag (all | configuration | connectivity );
    no-remote-trace;
  }
}
```

The attributes are as follows:

- **client** *client-id*—**outbound-ssh** configuration stanza on the device. Each **outbound-ssh** stanza represents a single outbound SSH connection. This attribute is not sent to the client.

  device-id *device-id*—Unique ID identifying the device running Junos OS to the configuration management server during the initiation process.

- **secret** *secret*—(Optional) Public SSH host key of the device. If this statement is added to the **outbound-ssh** configuration hierarchy, the device will pass its public key to the configuration management server during the initialization of the outbound SSH service. This is the recommended method of maintaining a current copy of the device's public key on the configuration management server.

- **keep-alive**—(Optional) Specify that keepalive messages be sent from the device running Junos OS to the configuration management server. To configure the keepalive message, you must set both the **timeout** and **retry** attributes.

  - **retry** *number*—Number of keepalive messages the device running Junos OS sends without receiving a response from the configuration management server before the current SSH connection is terminated. The default is three tries.

- **timeout *seconds***—Amount of time, in seconds, that the server waits for data before sending a keepalive signal. The default is 15 seconds.

- **reconnect-strategy (sticky | in-order)**—(Optional) Method that the device running Junos OS uses to reestablish a disconnected outbound SSH connection. Two methods are available:
  - **sticky**—The device attempts to reconnect to the configuration management server to which it was last connected. If the connection is unavailable, the device attempts to establish a connection with the next configuration management server on the list and so forth until a connection is established.
  - **in-order**—The device attempts to reestablish an outbound SSH session based on the configuration management server address list. The device attempts to establish a session with the first server on the list. If this connection is not available, the device attempts to establish a session with the next server, and so on down the list until a connection is established.

  When reconnecting to a client, the device running Junos OS attempts to reconnect to the client based on the **retry** and **timeout** values for each client listed in the configuration management server list.

- **services netconf**—(Required) Specifies the services available for the session. Currently, NETCONF is the only service available.

- **address**—(Required) The host name or the IPv4 address of the configuration management server. You can list multiple clients by adding each client's IP address or host name along with the connection parameters listed below:
  - **port *destination-port***—Outbound SSH port for the client. The default is port 22.
  - **retry *number***– Number of times the device running Junos OS attempts to establish an outbound SSH connection before gving up. The default is three tries.
  - **timeout *seconds***—Amount of time, in seconds, that the device running Junos OS attempts to establish an outbound SSH connection before giving up. The default is 15 seconds.

- **file *filename***—(Optional) Filename of the log file used to record the trace options. By default it is the name of the traced process (for example **mib2d** or **snmpd**). Use this option to override the default value.

- **files *files***—(Optional) Maximum number of trace files generated. By default, the maximum number of trace files is 10. Use this option to override the default value.

  When a trace file reaches its maximum size, the system archives the file and starts a new file. The system archives trace files by appending a number to the filename in sequential order from 1 to the maximum value (specified by the default value or the options value set here). Once the maximum value is reached, the numbering sequence is restarted at 1, overwriting the older file.

- **size *size***—(Optional) The maximum size of the trace file in kilobytes (KB). Once the maximum file size is reached, the system will archive the file. The default value is 1 MB. Use this option to override the default value.
- **match *match***—(Optional) Add lines to the trace file that match the the regular expression specified. For example, if the match value is set to **=error**, the system will only record lines to the trace file that include the string **error**.
- **(world-readable | no-world-readable)**—(Optional) This option specifies whether the files are accessible by the originator of the trace operation only or by any user. By default, log files are only accessible by the user that started the trace operation (**no-world-readable**). Use this option to override the default value.
- **(all | configuration | connectivity)**—(Optional) Flag specifying the type of tracing operation to perform.
  - **all**—Log all events.
  - **configuration**—Log all events pertaining to the configuration of the device running Junos OS.
  - **connectivity**—Log all events pertaining to the establishment of a connection between the client server and the device running Junos OS.
- **no-remote-trace**—(Optional) Disables remote tracing.

5. Commit the configuration:

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

Installing SSH Software on the Client

Once the device establishes the SSH connection to the configuration management server, the configuration management server takes control of the SSH session. Therefore, the SSH client software must be installed locally on the configuration management server.

If the client application accessing the NETCONF server uses the NETCONF Perl module provided by Juniper Networks, no further action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSH software. If the client application does not use theNETCONF Perl module, obtain the SSH client software and install it on the configuration management server where the application runs. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/ .

Receiving and Managing the Outbound SSH Initiation Sequence on the Client

When configured for outbound SSH, the device running Junos OS 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 within the device's configuration management server list. Prior to 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 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 public key is forwarded to the configuration management server by the device 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
```

Enabling NETCONF Service over SSH

RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, requires that the NETCONF server, by default, provide the client device with SSH access to the server over a dedicated TCP port. Use of a dedicated port makes it easy to identify and filter NETCONF traffic. The Internet Assigned Numbers Authority (IANA) port for NETCONF-over-SSH sessions is 830.

You also can enable access to the NETCONF server either over the default SSH port (22) or over a port number that is explicitly configured on the device running Junos OS. If you specify the NETCONF-over-SSH port number in the device configuration, the configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests. The default SSH port (22) continues to accept NETCONF sessions even when an alternate NETCONF-over-SSH port is configured. For added security, you can configure event policies that utilize **UI_LOGIN_EVENT** information to effectively disable the default port or further restrict NETCONF server access on a port.

To enable NETCONF service over SSH, perform the following steps:

1. Include one of the following statements at the indicated hierarchy level:
   - To enable SSH access over the default NETCONF-over-SSH port (830) as specified by RFC 4742, include the **ssh** statement at the **[edit system services netconf]** hierarchy level:

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

   - To enable SSH access over a specified port number, configure the **port** statement with the desired 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 NETCONF-over-SSH can be configured on any port from 1 through 65535, you should avoid configuring access on a port that is normally assigned for another service. This practice avoids potential resource conflicts. If you configure NETCONF-over-SSH on 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 to that effect.

- To enable access over the default SSH port (22), include the **ssh** statement at the **[edit system services]** hierarchy level. This configuration enables SSH access to the device for all users and applications. The **ssh** statement can be included in the configuration in addition to the NETCONF-over-SSH configuration statements listed above.

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

2. Commit the configuration:

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

3. Repeat the preceding steps on each device where the client application will establish a NETCONF session.

Connecting to the NETCONF Server

Before a client application can connect to the NETCONF server, you must satisfy the requirements described in "Prerequisites for Establishing an SSH Connection" on page 29.

When the prerequisites are satisfied, applications written in Perl use the NETCONF Perl module to connect to the NETCONF server. A client application that does not use the NETCONF Perl module uses one of two methods:

- It uses SSH library routines to establish an SSH connection to the NETCONF server, provide the username and password or passphrase, 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.
- It issues the following **ssh** command to create a NETCONF session as an SSH subsystem:

  ssh -p 830 -s *user*@*hostname* 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 in "Enabling NETCONF Service over SSH" on page 38.

  The **-s** option establishes the NETCONF session as an SSH subsystem.

The application must include code to intercept the NETCONF server's prompt for the password or passphrase. Perhaps the most straightforward method is for the application to use a utility such as the **expect** command. The NETCONF Perl client uses this method, for example.

Starting the NETCONF Session

Each NETCONF session begins with a handshake in which the NETCONF server and the client application specify the NETCONF capabilities they support. The following sections describe how to start a NETCONF session:

- Exchanging <hello> Tag Elements on page 40
- Verifying Compatibility on page 41

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 **<hello>** tag element encloses the **<capabilities>** tag element and the **<session-id>** tag element, which specifies the UNIX process ID (PID) of the NETCONF server for the session. Within the **<capabilities>** tag element, a **<capability>** tag element specifies each supported function.

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.

Each capability defined in the NETCONF specification is represented in a **<capability>** tag 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 XML management protocol for Junos OS Release 10.4 emits the following **<hello>** tag element (each **<capability>** tag element appears on three lines for legibility only):

```
<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:itf: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>
]]>]]>
```

(For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 28.)

The URIs in the **<hello>** tag element indicate the following supported capabilities:

- **urn:ietf:params:xml:ns:netconf:base:1.0**—The NETCONF server supports the basic NETCONF operations and tag elements defined in this namespace.
- **urn:ietf:params:xml:ns:netconf:capability:candidate:1.0**—The NETCONF server supports operations on a candidate configuration. For more information, see "Requesting Information from the Committed or Candidate Configuration" on page 63 and "Committing Configurations" on page 109.
- **urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0**—The NETCONF server supports confirmed commit operations. For more information, see "Committing the Candidate Configuration Only After Confirmation" on page 110.
- **urn:ietf:params:xml:ns: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 "Verifying a Configuration Before Committing It" on page 109.
- **urn:ietf:params:xml:ns: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 filesystem (indicated by the **file** option in the URN) and from remote machines by using Hypertext Transfer Protocol (HTTP) or FTP (indicated by the **http** and **ftp** options in the URN). For more information, see "Referencing Configuration Data Files" on page 87.
- **http://xml.juniper.net/netconf/junos/1.0**—The NETCONF server supports the operations defined in the Junos XML API for requesting and changing operational information (the tag elements in the *Junos XML API Operational Reference*). The NETCONF server also supports operations in the Junos XML management protocol for requesting or changing configuration information, but NETCONF client applications must use only native NETCONF XML management protocol operations for configuration functions. The semantics of corresponding Junos XML protocol operations and NETCONF XML protocol operations are not necessarily identical, so using Junos XML protocol configuration operations can lead to unexpected results.

To comply with the NETCONF specification, the client application also emits a **<hello>** tag element to define the capabilities it supports. It does not include the **<session-id>** tag element:

```
<hello>
<capabilities>
  <capability>first-capability</capability>
  <!-- tag elements for additional capabilities -->
</capabilities>
</hello>
]]>]]>
```

Verifying Compatibility

Exchanging **<hello>** tag elements enables a client application and the NETCONF server to determine if they support the same capabilities. In addition, we recommend that the

client application determine the version of the Junos OS running on the NETCONF server. After emitting its **<hello>** tag, the client application emits the **<get-software-information>** tag element in an **<rpc>** tag element:

```
<rpc>
  <get-software-information/>
</rpc>
]]>]]>
```

The NETCONF server returns the **<software-information>** tag element, which encloses the **<host-name>** and **<product-name>** tag elements plus a **<package-information>** tag element for each Junos OS module. (For information about the **<rpc-reply>** tag element, see "Parsing the NETCONF Server Response" on page 46.) The **<comment>** tag element within the **<package-information>** tag element specifies the Junos OS Release number (in the following example, **8.2** for Junos OS Release 8.2) and the build date in the form ***YYYYMMDD*** (year, month, day—12 January 2007 in the following example). Some tag elements appear on multiple lines, for legibility only:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" \
         xmlns:junos="http://xml.juniper.net/junos/8.2R1/junos">
  <software-information>
    <host-name>router1</host-name>
    <product-name>m20</product-name>
    <package-information>
      <name>junos</name>
      <comment>JUNOS Base OS boot [8.2-20070112.0]</comment>
    </package-information>
    <package-information>
      <name>jbase</name>
      <comment>JUNOS Base OS Software Suite \
              [8.2-20070112.0]</comment>
    </package-information>
    <!-- <package-information> tag elements for additional modules -->
  </software-information>
  </capabilities>
</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. Decide which of the following options is appropriate when there are differences, and implement the corresponding response:

- Ignore differences in capabilities and Junos version, and do not alter the client application's behavior to accommodate the NETCONF server. A difference in Junos 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 the Junos OS.
- End the NETCONF session and terminate the connection. This is appropriate if you decide that it is not practical to accommodate the NETCONF server's version or capabilities. For instructions, see "Ending a NETCONF Session and Closing the Connection" on page 53.

Exchanging Information with the NETCONF Server

The session continues when the client application sends a request to the NETCONF server. The NETCONF server does not emit any tag elements after session initialization except in response to the client application's requests. The following sections describe the exchange of tagged data:

Sending a Request to the NETCONF Server

To initiate a request to the NETCONF server, a client application emits the opening **<rpc>** tag, followed by one or more tag elements that represent the particular request, and the closing **</rpc>** tag, in that order:

```
<rpc>
    <!-- tag elements representing a request -->
</rpc>
]]>]]>
```

Each request is enclosed in its own separate pair of opening **<rpc>** and closing **</rpc>** tags and must constitute a well-formed XML document by including only compliant and correctly ordered tag elements. For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 28. For an example of emitting an **<rpc>** tag element in the context of a complete NETCONF session, see "Example of a NETCONF Session" on page 54.

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. For more information, see "Spaces, Newline Characters, and Other White Space" on page 13.

See the following sections for further information:

Request Classes

A client application can make three classes of requests:

- Operational Requests on page 44
- Configuration Information Requests on page 44
- Configuration Change Requests on page 45

NOTE: 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 element, so a client application can freely alternate operational and configuration requests.

Operational Requests

Operational requests are requests for information about the status of a device running Junos OS. Operational requests correspond to the CLI operational mode commands listed in the Junos OS command references. 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 sample request is for detailed information about the 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, see "Requesting Operational Information" on page 60. For information about the Junos XML request tag elements available in the current Junos OS Release, see the *Junos XML API Operational Reference*.

Configuration Information Requests

Configuration information requests are requests for information about the device's candidate configuration, a private configuration, or the committed configuration (the one currently in active use on the switching, routing, or security platform). 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 shows how to request 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, see "Requesting Configuration Information" on page 61. For a summary of the available configuration tag elements, see the *Junos XML API Configuration Reference*

Configuration Change Requests

Configuration change requests are requests to change the candidate configuration, or to commit those changes to put them into active use on the device running Junos OS. 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 CLI configuration statement described in the Junos OS configuration guides.

The following example shows how to create a new 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, see "Changing Configuration Information" on page 85. For a summary of Junos XML configuration tag elements, see the *Junos XML API Configuration Reference*

Including Attributes in the Opening <rpc> Tag

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.

Parsing the NETCONF Server Response

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. The opening **<rpc-reply>** tag includes the **xmlns** and **xmlns:junos** attributes (the opening tag appears here on multiple lines for legibility only):

```
<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 defines the namespace for enclosed tag elements that do not have the **junos:** prefix on their names and that are not enclosed in a child container tag that has the **xmlns** attribute with a different value.

The **xmlns:junos** attribute defines the namespace for enclosed Junos XML tag elements that have the **junos:** prefix on their names. The variable ***release*** is replaced by a code such as **10.4R1** for the initial version of Junos OS Release 10.4.

For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 28. For information about echoed attributes, see "Including Attributes in the Opening <rpc> Tag" on page 46.

Client applications must include code for parsing the stream of response tag elements coming from the NETCONF server, either processing them as they arrive or storing them until the response is complete. See the following sections for further information:

- NETCONF Server Response Classes on page 47
- Using a Standard API to Parse Response Tag Elements on page 48

NETCONF Server Response Classes

The NETCONF server returns three classes of responses:

Operational Responses

Operational responses are responses to requests for information about the status of a switching, routing, or security platform. They correspond to the output from CLI operational commands as described in the Junos command references.

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 element in a response tag element called **<interface-information>** and returns the information requested by the **<get-chassis-inventory>** tag element in a response tag element called **<chassis-inventory>**.

The following sample response includes information about the interface **ge-2/3/0**. The namespace indicated by the **xmlns** attribute in the opening **<interface-information>** tag is for interface information in the initial version of Junos OS Release 10.4. The opening tags appear on two lines here for legibility only:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
        xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
  <interface-information \
        xmlns="http://xml.juniper.net/junos/10.4R1/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 "Requesting Operational Information" on page 60. For a summary of operational response tag elements, see the *Junos XML API Operational Reference*.

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 the information at the **[edit system login]** hierarchy level in the configuration hierarchy. For brevity, the sample shows only one user defined at this level. The opening **<rpc-reply>** tag appears on two lines for legibility only. For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
  xmlns:junos="http://xml.juniper.net/junos/10.4R1/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>
]]>]]>
```

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>** tag element that describes the cause of the failure. For information about handling errors, see "Handling an Error or Warning" on page 49.

For information about changing the device configuration, see "Changing Configuration Information" on page 85. For a summary of the available configuration tag elements, see the *Junos XML API Configuration Reference*.

Using a Standard API to Parse Response Tag Elements

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 document.

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 http://search.cpan.org/~tjmather/XML-DOM/lib/XML/DOM.pm .

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/ .

Handling an Error or Warning

If the NETCONF server encounters an error condition, it emits an **<rpc-error>** tag element containing tag 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>
]]>]]>
```

<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.

<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**.

An error can occur while the server is performing any of the following operations, and the server can send a different combination of child tag elements in each case:

- Processing an operational request submitted by a client application
- 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>** tag 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>** tag 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.

Locking and Unlocking the Candidate Configuration

When a client application is requesting or changing configuration information, it can use one of two methods to access the configuration:

- Lock the candidate configuration, which prevents other users or applications from changing it until the application releases the lock (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 configuration 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, as described in "Requesting Configuration Information" on page 61. However, it is appropriate to lock the configuration if it is important that the information being returned not change during the session.

For more information about locking and unlocking the candidate configuration, see the following sections:

- Locking the Candidate Configuration on page 50
- Unlocking the Candidate Configuration on page 51

Locking the Candidate Configuration

To lock the candidate configuration, a client application emits the **<lock>** and **<target>** tag elements and the **<candidate/>** tag in the **<rpc>** tag element:

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
]]>]]>
```

Locking the candidate configuration prevents other users or applications from changing the candidate configuration until the lock is released (equivalent to the CLI **configure exclusive** command). Locking the configuration before making changes is recommended, 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.

The NETCONF server confirms that it has locked the candidate by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot lock the configuration, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag 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. If the client application has the necessary Junos access privilege, it can terminate the session that holds the lock. For more information, see "Terminating Another NETCONF Session" on page 52.
- The candidate configuration already includes changes that have not yet been committed. To commit the changes, see "Committing a Configuration" on page 110. To discard uncommitted changes, see "Rolling Back a Configuration" on page 97.

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 NETCONF session ends or the client application unlocks the configuration by emitting the **<unlock>** tag element, as described in "Unlocking the Candidate Configuration" on page 51.

If the candidate configuration is not committed before the client application unlocks it, 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 includes the **<unlock>** and **<target>** tag elements and the **<candidate/>** tag in an **<rpc>** tag element:

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

The NETCONF server confirms that it has unlocked the candidate by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot unlock the configuration, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag element explaining the reason for the failure.

Terminating Another NETCONF Session

A client application's attempt to lock the candidate configuration can fail because another user or application already holds the lock, as mentioned in "Locking the Candidate Configuration" on page 50. 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 **maintenance** permission, it can end the session that holds the lock by emitting the **<kill-session>** and **<session-id>** tag elements in an **<rpc>** tag element. The **<session-id>** tag 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 has terminated the other session by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<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, based on 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. (For information about the confirmed commit operation, see "Committing the Candidate Configuration Only After Confirmation" on page 110.)

The following example shows how to terminate another session:

Client Application

```
<rpc>
  <kill-session>
    <session-id>3250</session-id>
  </kill-session>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2101

Ending a NETCONF Session and Closing 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>** tag element:

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

In response, the NETCONF server emits the **<ok/>** tag enclosed in an **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

For an example of the exchange of closing tag elements, see "Closing the NETCONF Session" on page 58.

Because the connection to the NETCONF server is an SSH subsystem, it closes automatically when the NETCONF session ends.

Displaying CLI Output as XML Tag Elements

To display the output from a CLI command as NETCONF and Junos XML tag elements instead of as the default formatted ASCII, pipe the command to the **display xml** command. Infrastructure tag elements in the response belong to the Junos XML management protocol instead of the NETCONF XML management protocol. The tag elements that describe Junos OS configuration or operational data belong to the Junos XML API, which defines the Junos content that can be retrieved and manipulated by both the Junos XML management protocol and the NETCONF XML management protocol operations.

The following example shows the output from the **show chassis hardware** command issued on an M20 router that is running the initial version of Junos OS Release 10.4 (the opening **<chassis-inventory>** tag appears on two lines for legibility only):

```
user@host> show chassis hardware | display xml
  <rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
```

```
    <chassis-inventory \
          xmlns="http://xml.juniper.net/junos/10.4R1/junos-chassis">
      <chassis junos:style="inventory">
        <name>Chassis</name>
        <serial-number>00118</serial-number>
        <description>M20</description>
        <chassis-module>
          <name>Backplane</name>
          <version>REV 06</version>
          <part-number>710-001517</part-number>
          <serial-number>AB5911</serial-number>
        </chassis-module>
        <chassis-module>
          <name>Power Supply A</name>
          <!-- other child tags of <chassis-module> -->
        </chassis-module>
        <!-- other child tags of <chassis> -->
      </chassis>
    </chassis-inventory>
  </rpc-reply>
  ]]>]]>
```

Displaying the RPC Tags for a Command

To display the remote procedure call (RPC) XML tags for an operational mode command, enter **display xml rpc** after the pipe symbol (|).

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/10.1I0/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Example of a NETCONF Session

This section describes the sequence of tag elements in a sample NETCONF session. The client application begins by establishing a connection to a NETCONF server. See the following sections:

- Exchanging Initialization Tag Elements on page 55
- Sending an Operational Request on page 55
- Locking the Configuration on page 56
- Changing the Configuration on page 56
- Committing the Configuration on page 57

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 "Generating Well-Formed XML Documents" on page 28. For a detailed discussion of the **<hello>** tag element, see "Exchanging <hello> Tag Elements" on page 40.

NETCONF Client Application
Server

```
<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>
]]>]]>
```

T2102

Sending an Operational Request

The client application now 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

```
<rpc>
  <get-chassis-inventory>
    <detail/>
  </get-chassis-inventory>
</rpc>
]]>]]>
```

NETCONF Server

```
<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>
]]>]]>
```

T2103

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. For more information and locking the configuration, see "Locking the Candidate Configuration" on page 50.

Client Application

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2104

Changing the Configuration

The client application now emits tag elements to create a new Junos login class called **network-mgmt** at the **[edit system login class]** hierarchy level in the candidate configuration. To confirm that it incorporated the changes, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. (Understanding the meaning of these tag elements is not necessary for the purposes of this example, but for information about them, see "Changing Configuration Information" on page 85.)

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>
]]>]]>
```

T2105

Committing the Configuration

The client application commits the candidate configuration. To confirm that it committed the candidate configuration, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. For more information about the commit operation, see "Committing the Candidate Configuration" on page 110.

Client Application **NETCONF Server**

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2106

Unlocking the Configuration

The client application unlocks (and by implication closes) the candidate configuration. To confirm that it unlocked the candidate configuration, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. For more information about unlocking a configuration, see "Unlocking the Candidate Configuration" on page 51.

Client Application

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2107

Closing the NETCONF Session

The client application closes the NETCONF session. For more information about closing the session, see "Ending a NETCONF Session and Closing the Connection" on page 53.

Client Application

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

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2108

CHAPTER 4

Requesting Information

This chapter explains how to use the NETCONF XML management protocol and Junos XML API to request information about the status and the current configuration of a routing, switching, or security platform.

The tag elements for operational requests are defined in the Junos XML API and correspond to Junos OS command-line interface (CLI) operational commands, which are described in the Junos OS command references. 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, which is described in the *Junos OS CLI User Guide*. 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 an XML schema representation of the configuration hierarchy, information about previously committed (rollback) configurations, or information about the rescue configuration.

This chapter discusses the following topics:

Overview of the Request Procedure

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, as described in "Connecting to the NETCONF Server" on page 39.
2. Opens a NETCONF session, as described in "Starting the NETCONF Session" on page 40.
3. If making configuration requests, optionally locks the candidate configuration, as described in "Locking the Candidate Configuration" on page 50.
4. Makes any number of requests one at a time, freely intermingling operational and configuration requests. See "Requesting Operational Information" on page 60 and "Requesting Configuration Information" on page 61.

   The application can also intermix requests with configuration changes, which are described in "Changing Configuration Information" on page 85.
5. Accepts the tag stream emitted by the NETCONF server in response to each request and extracts its content, as described in "Parsing the NETCONF Server Response" on page 46.
6. Unlocks the candidate configuration if it is locked, as described in "Unlocking the Candidate Configuration" on page 51. Other users and applications cannot change the configuration while it remains locked.
7. Ends the NETCONF session and closes the connection to the device, as described in "Ending a NETCONF Session and Closing the Connection" on page 53.

Requesting Operational Information

To request information about the current status of a device, a client application emits the specific tag element from the Junos XML API that returns the desired information. For example, the **<get-interface-information>** tag element corresponds to the **show interfaces** command, the **<get-chassis-inventory>** tag element requests the same information as the **show chassis hardware** command, and the **<get-system-inventory>** tag element requests the same information as the **show software information** command.

For complete information about the operational request tag elements available in the current Junos OS Release, see the chapters in the *Junos XML API Operational Reference* that are titled "Mapping Between Operational Tag Elements, Perl Methods, and CLI Commands" and "Summary of Operational Request Tag Elements."

The application encloses the request tag element in an **<rpc>** tag element. The syntax depends on whether the corresponding CLI command has any options:

```
<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 NETCONF server encloses its response in the specific response tag element that corresponds to the request tag element, 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>
]]>]]>
```

The opening tag for each operational response includes the **xmlns** attribute to define 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. 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*

release-code is the standard string that represents the release of the Junos OS running on the NETCONF server device.

category specifies the DTD.

The *Junos XML API Operational Reference* includes the text of the Junos XML DTDs for operational responses.

Parsing the <output> Tag Element

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 encloses its response in an **<output>** tag element. The tag element's contents are usually one or more lines of formatted ASCII output like that displayed by the CLI on the computer screen.

NOTE: The content and formatting of data within an <output> tag element are subject to change, so client applications must not depend on them. Future versions of the Junos XML API will define specific response tag elements (instead of <output> tag elements) for more commands. Client applications that rely on the content of <output> tag elements will not be able to interpret the output from future versions of the Junos XML API.

Requesting Configuration Information

To request information about a configuration on 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 either the candidate or active configuration. 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 -->
    </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 the syntax used within the **<filter>** tag element to represent elements, see "Specifying the Scope of Configuration Information to Return" on page 65.

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. For more information, see "Locking and Unlocking the Candidate Configuration" on page 50.

The NETCONF server encloses its reply in **<configuration>**, **<data>**, and **<rpc-reply>** 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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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.

Client applications can also request other configuration-related information, including an XML schema representation of the configuration hierarchy or information about previously committed configurations. For more information, see the following sections:

- Requesting an XML Schema for the Configuration Hierarchy on page 76
- Requesting a Previous (Rollback) Configuration on page 80
- Comparing Two Previous (Rollback) Configurations on page 81
- Requesting the Rescue Configuration on page 83

The following sections describe how a client application specifies the source and scope of configuration information returned by the NETCONF server:

- Requesting Information from the Committed or Candidate Configuration on page 63
- Specifying the Scope of Configuration Information to Return on page 65

Requesting Information from the Committed or Candidate Configuration

To request information from the candidate configuration, a client application includes the **<source>** tag 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>
]]>]]>
```

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 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 requesting the entire configuration, the application omits the <filter> tag element. For information about the <filter> tag element, see "Specifying the Scope of Configuration Information to Return" on page 65.

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, the NETCONF server also 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 also 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.

Specifying the Scope of Configuration Information to Return

By including the appropriate child tag elements in the **<filter>** tag element within the **<rpc>** and **<get-config>** tag elements, a client application can request the entire configuration or specific portions of the configuration:

```
<rpc>
  <get-config>
    <source>
      ( <candidate/> | <running/> )
    </source>
    <filter>
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about requesting different amounts of configuration information, see the following sections:

- Requesting the Complete Configuration on page 65
- Requesting a Hierarchy Level or Container Object Without an Identifier on page 66
- Requesting All Configuration Objects of a Specified Type on page 67
- Requesting Identifiers for Configuration Objects of a Specified Type on page 69
- Requesting One Configuration Object on page 71
- Requesting Specific Child Tags for a Configuration Object on page 73
- Requesting Multiple Configuration Elements Simultaneously on page 75

Requesting the Complete Configuration

To request the entire candidate configuration, 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>
]]>]]>
```

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 **<configuration>**, **<data>**, and **<rpc-reply>** tag elements. For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- Junos XML tag elements representing the configuration -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

Requesting a Hierarchy Level or Container Object Without an Identifier

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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 "Requesting Multiple Configuration Elements Simultaneously" on page 75.

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>
]]>]]>
```

T2128

Requesting All Configuration Objects of a Specified Type

To request complete 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 represents the requested object type. 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/>
        <!-- closing tags for each parent of the requested object type -->
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Requesting Information from the Committed or Candidate Configuration" on page 63.

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 "Requesting a Hierarchy Level or Container Object Without an Identifier" on page 66.

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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 "Requesting Multiple Configuration Elements Simultaneously" on page 75.

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>$9$Pf3900REcr/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>
]]>]]>
```

T21129

Requesting Identifiers for Configuration Objects of a Specified Type

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

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 "Requesting All Configuration Objects of a Specified Type" on page 67.

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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 "Requesting Multiple Configuration Elements Simultaneously" on page 75.

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>
]]>]]>
```

T2130

Requesting One Configuration Object

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 its container tag element and each of its identifier tag elements, complete with identifier value. 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 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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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 "Requesting Multiple Configuration Elements Simultaneously" on page 75.

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>
]]>]]>
```

T2131

Requesting Specific Child Tags for a Configuration Object

To request specific child tag elements for 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 its container tag element and 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. To represent the child tag elements to return, it emits each one as 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 object -->
          <object>
            <name>identifier</name>
            <first-child/>
            <second-child/>
            <!-- empty 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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

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 "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- opening tags for each parent of the object -->
        <object>
          <name>identifier</name>
        </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 "Requesting Multiple Configuration Elements Simultaneously" on page 75.

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>
]]>]]>
```

T2132

Requesting Multiple Configuration Elements Simultaneously

Within a **<get-config>** tag element, a client application can request multiple configuration elements of the same type or different types. 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:

- Requesting a Hierarchy Level or Container Object Without an Identifier on page 66
- Requesting All Configuration Objects of a Specified Type on page 67
- Requesting Identifiers for Configuration Objects of a Specified Type on page 69
- Requesting One Configuration Object on page 71
- Requesting Specific Child Tags for a Configuration Object on page 73

Requesting an XML Schema for the Configuration Hierarchy

To request an XML Schema-language representation of the entire configuration hierarchy, a client application emits the Junos XML **<get-xnm-information>** tag element and its **<type>** and **<namespace>** child tag elements with the indicated values in an **<rpc>** tag element:

```
<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>** tag elements:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <xsd:schema>
    <!-- tag elements for the Junos XML schema -->
  </xsd:schema>
</rpc-reply>
]]>]]>
```

The schema represents all configuration elements available in the version of the Junos OS that is running on a device. (To determine the Junos OS version, emit the **<get-software-information>** operational request tag element, which is documented in the *Junos XML API Operational Reference*.)

Client applications can use the schema to validate the configuration on a device or simply to learn which configuration statements are available in the version of the Junos OS running on the device. The schema does not indicate which elements are actually configured or even that an element can be configured 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, as described in "Requesting Configuration Information" on page 61.

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*, available from the World Wide Web Consortium (W3C) at http://www.w3.org/TR/xmlschema-0/ . The primer provides a basic introduction and lists the formal specifications where you can find detailed information.

For further information, see the following sections:

- Creating the junos.xsd File on page 77
- Example: Requesting an XML Schema on page 78

Creating 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.

In Junos OS Release 6.4 and later, the **<xsd:import>** tag element is enclosed in the **<xsd:schema>** tag element and references the file **junos.xsd**, which contains the required information about the **junos** namespace. For example, the following **<xsd:import>** tag

element specifies the file for Junos OS Release 10.4R1 (and appears on two lines for legibility only):

```
<xsd:import schemaLocation="junos.xsd" \
    namespace="http://xml.juniper.net/junos/10.4R1/junos"/>
```

To enable the schema validator to interpret the **<xsd:import>** tag element, you must manually create a file called **junos.xsd** in the directory where you place the **.xsd** file that contains the complete Junos configuration schema. 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 running on the device (for example, **10.4R1** for the first release of Junos OS 10.4).

```
<?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: Requesting an XML Schema

The following examples show how to request the Junos configuration schema. In the NETCONF server's response, the first **<xsd:element>** statement defines the **<undocumented>** Junos XML tag element, which 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 NETCONF server's response appear on multiple lines for legibility only. The NETCONF server does not insert newline characters within tags or tag elements. Also, in actual output the ***JUNOS-version*** variable is replaced by a value such as **10.4R1** for the initial version of Junos OS Release 10.4.

Client Application **NETCONF Server**

```
<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>
                                      .
                                      .
                                      .
```

T2114

Another **<xsd:element>** statement near the beginning of the schema defines the Junos XML **<configuration>** tag element. It encloses the **<xsd:element>** statement that defines the **<system>** tag element, which corresponds to the **[edit system]** hierarchy level. The statements corresponding to other hierarchy levels are omitted for brevity.

Client Application **NETCONF Server**

```
                                      .
                                      .
                                      .
                            </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>
                        ]]>]]>
```

T2115

Requesting a Previous (Rollback) Configuration

To request a previously committed (rollback) configuration, a client application emits the Junos XML **<get-rollback-information>** tag element and its child **<rollback>** tag element in an **<rpc>** tag element. This operation is equivalent to the **show system rollback** operational mode command. The **<rollback>** tag element specifies the index number of the previous configuration to display; its value can be from **0** (zero, for the most recently committed configuration) through **49**.

To request Junos XML-tagged output, the application either includes the **<format>** tag element with the value **xml** or omits the **<format>** tag element (Junos XML tag elements are the default):

```
<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>** tag elements. The **<ok/>** tag is a side effect of the implementation and does not affect the results. For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rollback-information>
    <ok/>
    <configuration attributes>
      <!-- tag elements representing the complete previous configuration -->
    </configuration>
  </rollback-information>
</rpc-reply>
]]>]]>
```

To request formatted ASCII output, the application includes the **<format>** tag element with the value **text**:

```
<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>** tag elements. For more information about the formatted ASCII notation used in Junos configuration statements, see the *Junos OS CLI User Guide*.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rollback-information>
```



```
      <ok/>
      <configuration-information>
        <configuration-output>
          /* formatted ASCII representing the complete previous configuration*/
        </configuration-output>
      </configuration-information>
    </rollback-information>
  </rpc-reply>
  ]]>]]>
```

The following example shows how to request Junos XML-tagged output for the rollback configuration that has an index of **2**. In actual output, the ***Junos-version*** variable is replaced by a value such as **10.4R1** for the initial version of Junos OS Release 10.4.

Client Application

```
<rpc>
  <get-rollback-information>
    <rollback>2</rollback>
  </get-rollback-information>
</rpc>
]]>]]>
```

NETCONF Server

```
<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>
]]>]]>
```

T2133

Comparing Two Previous (Rollback) Configurations

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

Requesting the Rescue Configuration

To request the rescue configuration, a client application emits the Junos XML **<get-rescue-information>** tag element in an **<rpc>** tag element. This operation is equivalent to the **show system configuration rescue** operational mode command.

The rescue configuration is a configuration saved in case it is necessary to restore a valid, nondefault configuration. (To create a rescue configuration, use the Junos XML **<request-save-rescue-configuration>** tag element or the **request system configuration rescue save** CLI operational mode command. For more information, see the *Junos XML API Operational Reference* or the *Junos OS System Basics and Services Command Reference*.)

To request Junos XML-tagged output, the application either includes the **<format>** tag element with the value **xml** or omits the **<format>** tag element (Junos XML tag elements are the default):

```
<rpc>
  <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. For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 63.

```
<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 formatted ASCII output, the application includes the **<format>** tag 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. For more information about the formatted ASCII notation used in Junos configuration statements, see the *Junos OS CLI User Guide*.

```
<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>
]]>]]>
```


CHAPTER 5

Changing Configuration Information

This chapter explains how to use the NETCONF XML management protocol along with Junos XML or command-line interface (CLI) configuration statements to change the configuration on a routing, switching, or security platform. The NETCONF protocol operations**<copy-config>**, **<edit-config>**, and **<discard-changes>** offer functionality that is analogous to configuration mode commands in the Junos OS CLI. These CLI configuration mode commands, as well as the CLI configuration statements, are described in the *Junos OS CLI User Guide*. The Junos XML tag elements described here correspond to configuration statements, which are described in the Junos OS configuration guides.

This chapter discusses how to use the NETCONF XML management protocol to make changes to a device's configuration. To see how this activity fits in within the overall NETCONF session, see "Client Application's Role in a NETCONF Session" on page 27.

This chapter discusses the following topics:

Editing the Candidate Configuration

To change the candidate configuration on a device, 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 shows the various tag elements available:

```
<rpc>
  <copy-config>
    <target><candidate/></target>
    <default-operation> (merge | none | replace) </default-operation>
    <error-operation> (ignore-error | stop-on-error) </error-operation>
    <source><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-test | url)>
   </edit-config>
</rpc>
]]>]]>
```

```
<rpc>
  <discard-changes/>
</rpc>
]]>]]>
```

Notice that the three tags—**<copy-config>**, **<edit-config>**, and **<discard-changes>**—correspond to the three basic configuration tasks available to you:

- Overwriting the candidate configuration with a new configuration—Using the **<copy-config>** tag element, you can replace the current candidate configuration with a new configuration.
- Editing the candidate configuration elements—Using the **<edit-config>** tag element, you can add, change, or delete specific configuration elements within the candidate configuration.
- Rolling back changes to the current configuration—Using the **<discard-changes>** tag element, you can roll back the candidate configuration to a previously committed configuration. This tag element provides functionality analogous to the CLI command **rollback**.

Notice also that the **<copy-config>** and the **<edit-config>** tags both have additional subtags related to each tag element. These subtag elements are described in the following sections:

Formatting the Configuration Data

A client application can use a text file or streaming data to deliver configuration data to the candidate configuration. The data delivered can be in one of two formats: Junos XML or CLI configuration statements. You can specify the delivery mechanism and the format used when delivering configuration changes to the device.

For more information on Junos XML tag elements, see "XML Overview" on page 6. For more information on CLI configuration statements, see the *Junos OS CLI User Guide*.

Delivery Mechanism: Data Files Versus Streaming Data

When formatting your configuration data output, you can choose to stream your configuration changes within your 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; with data files, there is no need to include the possibly complex configuration data stream.

Referencing Configuration Data Files

To reference configuration data as a file, a client application emits the file location between **<url>** tag elements within the **<rpc>** and the **<edit-config>** tag elements.

```
<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 or CLI configuration statements. When the configuration data is formatted as CLI configuration statements, you set the **<url>** format attribute to **text**.

```
<rpc>
  <edit-config>
    ...
    <url format="text">
     <!-- location and name of file containing configuration data -->
    </url>
  </edit-config>
</rpc>
```

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 based on the user's home directory.
  - 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.
- When located on the network, the configuration file can be accessed using FTP or HTTP:
  - FTP example:

    ftp://*username*:*password*@*hostname*/*path*/*filename*

    

    NOTE: The default value for the FTP ***path*** variable is the user's home directory. Thus, by default the file path to the configuration file is relative to the user directory. To specify an absolute path when using FTP, start the path with the characters %2F; for example: ftp://*username*:*password*@*hostname*/%2F*path*/*filename*.
  - HTTP example:

    http://*username*:*password*@*hostname*/*path*/*filename*

Before loading the file, the client application or an administrator saves Junos XML tag elements as the contents of the file. The file includes the tag elements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to each element to change. The notation is the same as that used to request configuration information, as described in "Requesting Information" on page 59. For more detailed information about the Junos XML representation of Junos configuration statements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16.

The following example shows how to incorporate configuration data stored in the file **/var/tmp/configFile** on the FTP server called **ftp.myco.com**:

Client Application **NETCONF Server**

```
<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>
]]>]]>
```

T2134

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. The Junos XML notation is the same as that used to request configuration information, as described in "Requesting Information" on page 59. For more detailed information about the mappings between Junos configuration elements and Junos XML tag elements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16. The CLI configuration statement notation are further described in the *CLI User Guide*.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config> or <config-text>
      <configuration> or <configuration-text>
        <!-- configuration changes -->
      </configuration> or </configuration-text>
    </config> or </config-text>
  </edit-config>
</rpc>
]]>]]>
```

The following example shows how to provide Junos XML configuration data for the **messages** system log file in a data stream:

Client Application **NETCONF Server**

```
<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>
]]>]]>
```

T2135

Data Format: Junos XML versus CLI Configuration Statements

You can format the configuration data using one of two formats: Junos XML or CLI configuration statements. The choice between one data format over the other is personal preference.

If you are supplying the configuration changes in the form of data files, you enclose the data filename and path within **<url>** tags. By default, these tags specify that the referenced data files are written in Junos XML. Thus, the following code declares that the data within the file is Junos XML:

```
<url>dataFile</url>
```

To specify that the data file be written as CLI configuration statements, you set the **<url>** tag's format attribute to **text**:

```
<url format=”text”>dataFile</url>
```

When streaming data, you specify the data format by selecting one of two tags: **<config>** for Junos XML statements and **<config-text>** for CLI configuration statements.

In the following example, Junos XML formatted configuration data is included between the **<configuration>** tag:

```
<config>
 <configuration>
  <system>
   <services>
    <ssh>
     <protocol-version>v2</protocol-version>
    </ssh>
   </services>
  </system>
 <configuration>
</config>
```

In this next example, the same data written formatted as CLI configuration statements and included within **<configuration-text>** tags:

```
<config-text>
 <configuration-text>
  system {
   services {
    ssh {
     protocol-version v2 ;
    }
   }
  }
 </configuration-text>
</config-text>
```

Setting the Edit Configuration Mode

When sending operation data to the NETCONF server, you have the option to specify how a device should handle these 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 only for specific elements within the session.

The device has the following edit configuration modes:

- **merge**—The device merges new configuration data into the current candidate configuration. This is the default.
- **replace**—The device replaces existing configuration data with the new configuration data.
- **no-change**—The device does not change the existing configuration unless the new configuration element includes an operation attribute.

To set the mode globally for the session, place a configuration mode value within **<default-operation>** tags:

```
<rpc>
 <edit-config>
  <default-operation>ConfigModeValue</default-operation>
 <edit-config>
</rpc>
```

You can also set the mode for a specific configuration statement by adding an **operation** attribute with a value of **replace** to the configuration element:

```
<rpc>
 <edit-config>
  <config>
   <configuration>
    <protocols>
     <rip>
      <message-size operation=”replace”>255</message-size>
     </rip>
    </protocols>
   </configuration>
  </config>
 </edit-config>
</rpc>
```

You can 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>
```

Specifying the merge Data Mode

By default, the NETCONF server *merges* new configuration data into the candidate configuration. Thus, if no edit-configuration mode is specified, the device will merge the new configuration elements into the existing candidate configuration. Merging configurations is performed according to the following rules:

- 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 can include 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 candidate configuration. To specify that the data be replaced, the application can include 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 candidate configuration with new configuration data. Furthermore, when the edit configuration mode is set to **replace**, we do not recommend using the **operation** attribute on individual configuration elements.

You can also replace individual configuration elements while merging or creating others. See "Replacing Configuration Elements" on page 100.

Specifying the no-change Data Mode

In the *no-change* mode, configuration 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 "Deleting Configuration Elements" on page 103:

To set the no-change edit configuration mode globally, the application can include 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 does not exist in the candidate, the NETCONF server returns an error. We recommend using no-change mode only when removing configuration elements from the candidate configuration. When creating or modifying elements, applications need to use merge mode. For more information, see "Deleting Configuration Elements" on page 103.

When the no-change edit configuration mode is set globally, using the **<default-operation>** tag, you can override this behavior by specifying a different edit configuration mode for a specific element using the **operation** attribute. 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>
```

Handling Errors

If the NETCONF server cannot incorporate the configuration data, the **<rpc-error>** tag element is returned 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. A client application can explicitly specify this response to errors by including 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. For information about the **<commit>** and **<validate>** tag elements, see "Committing Configurations" on page 109.

Replacing the Candidate Configuration

You can replace the candidate configuration with a new configuration file using the **<copy-config>** tag, or you can use the **<edit-config>** tag with the **<default-operation>** subtag value set to **replace**.

For information about completely replacing the candidate configuration, see the following sections:

- Using <copy-config> on page 96
- Using <edit-config> on page 96

Using <copy-config>

One method for replacing the entire candidate configuration is to include the **<copy-config>** tag element in the **<rpc>** tag element. The **<source>** tag element encloses the **<url>** tag element to specify the filename that contains the new configuration data. The **<target>** tag element encloses the **<candidate/>** tag to indicate that the new configuration data replaces the candidate configuration:

```
<rpc>
  <copy-config>
    <target>
      <candidate/>
    </target>
    <source>
      <url>
        <!-- location specifier for file containing the new configuration -->
      </url>
    </source>
  </copy-config>
</rpc>
]]>]]>
```

Using <edit-config>

The other method for replacing the entire candidate configuration is to set the edit configuration mode to **replace** as a global variable. The candidate configuration includes the **<default-operation>** tag element with the value **replace** in the **<edit-config>** tag element, as described in "Setting the Edit Configuration Mode" on page 91. To specify the new configuration data, the application includes either a **<config>** tag element that contains the data or a **<url>** tag element that names the file containing the data, as discussed in "Formatting the Configuration Data" on page 87.

```
<rpc>
  <edit-config>
    <default-operation>replace</default-operation>
    <source>

    <!-- EITHER -->
      <config>
        <!-- tag elements representing the new configuration -->
      </config>
    <!-- OR -->
      <url>
        <!-- location specifier for file containing the new configuration -->
      </url>

    </source>
  </edit-config>
</rpc>
]]>]]>
```

Rolling Back a Configuration

The **<discard-changes>** tag allows you to roll back the candidate configuration to a previous configuration. To roll back the candidate to the current running configuration, insert the **<discard-changes>** tag within the **<rpc>** tag.

```
<rpc>
  <discard-changes/>
</rpc>
]]>]]>
```

This operation is equivalent to the CLI configuration mode **rollback 0** command.

The NETCONF server indicates that it discarded the changes by returning the **<load-success/>** tag after you issue the **</discard-changes>** tag.

Deleting the Candidate Configuration

You can use the **<delete-config>** tag element to delete the current candidate configuration. Exercise caution when issuing the **<delete-config>** tag element. If you commit an empty candidate configuration, the device will go offline.

```
<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 on the console port, see the hardware manual for your specific device.

Changing Individual Configuration Elements

You change individual configuration elements within a candidate configuration using the **<edit-config>** tag element within the **<rpc>** tag. By default, the NETCONF server merges new configuration data into the existing candidate 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:

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>

  <!-- EITHER -->
    <config>
      <configuration>
        <!-- tag elements representing the configuration elements to change -->
      </configuration>
    </config>
  <!-- OR -->
    <config-text>
      <configuration-text>
        <!-- tag elements representing the configuration elements to change -->
      </configuration-text>
    </config-text>
  <!-- OR -->
    <url>
      <!-- location specifier for file containing changes -->
    </url>

  </edit-config>
</rpc>
]]>]]>
```

Using configuration data within the **<config>** or **<config-text>** tag elements or the file specified within the **<url>** tag element, the application defines a configuration element by including 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 element. To represent the element, the application includes its container tag element. The child tag elements included within the container tag element depend on the operation, and are described in the following sections:

- Merging Configuration Elements on page 99
- Replacing Configuration Elements on page 100
- Creating New Configuration Elements on page 101
- Deleting Configuration Elements on page 103

For more information about the tag elements that represent configuration statements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16. For

information about the tag elements for a specific configuration element, see the *Junos XML API Configuration 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>
]]>]]>
```

For more information, see the following sections:

Merging Configuration Elements

To merge configuration elements (hierarchy levels or configuration objects) into the candidate configuration, a client application emits the basic tag elements described in "Changing Individual Configuration Elements" on page 98.

To represent each element to merge in (either within the **<config>** tag element or in the file named by the **<url>** tag element), the application includes the tag elements representing its parent hierarchy levels and its container tag element, as described in "Changing Individual Configuration Elements" on page 98. 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 "Setting the Edit Configuration Mode" on page 91. 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 **NETCONF Server**

```
<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>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2120

Replacing Configuration Elements

To replace configuration elements (hierarchy levels or configuration objects) in the candidate configuration, a client application emits the basic tag elements described in "Changing Individual Configuration Elements" on page 98.

To represent the new definition for each configuration element being replaced (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 and its container tag element, as described in "Changing Individual Configuration Elements" on page 98. 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, 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.

The application can also replace all objects in the configuration in one operation. For instructions, see "Replacing the Candidate Configuration" on page 95.

The following example shows how to grant new permissions for the object named **operator** at the **[edit system login class]** hierarchy level.

Client Application

```
<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>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2121

Creating New Configuration Elements

To create configuration elements (hierarchy levels or configuration objects) in the candidate configuration only if the elements do not already exist, a client application emits the basic tag elements described in "Changing Individual Configuration Elements" on page 98.

To represent each configuration element being created (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 and its container tag element, as described in "Changing Individual Configuration Elements" on page 98. 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 the 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 candidate 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:

Client Application

```
<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>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2122

Deleting Configuration Elements

To delete a configuration element (hierarchy level or configuration object) from the candidate configuration, a client application emits the basic tag elements described in "Changing Individual Configuration Elements" on page 98. 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 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 the only operation performed in no-change mode be deletion. When merging, replacing, or creating configuration elements, client applications use merge mode.

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 "Changing Individual Configuration Elements" on page 98. The tag element in which the **operation="delete"** attribute is included depends on the element type, as described in the following sections:

- Deleting a Hierarchy Level or Container Object on page 104
- Deleting a Configuration Object That Has an Identifier on page 104
- Deleting a Single-Value or Fixed-Form Option from a Configuration Object on page 105
- Deleting Values from a Multivalue Option of a Configuration Object on page 106

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 in the empty tag that represents the level:

```
<configuration>
  <!-- opening tags for each parent level -->
    <level-to-delete operation="delete"/>
  <!-- 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 "Deleting Configuration Elements" on page 103. For more information about hierarchy levels and container objects, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16.

The following example shows how to remove the **[edit protocols ospf]** hierarchy level of the candidate configuration:

Client Application

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <protocols>
          <ospf operation="delete"/>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2123

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 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">
      <name>identifier</name>
```

```
    </object>
  <!-- closing tags for each parent of the object -->
</configuration>
```

NOTE: The delete attribute appears in the opening container tag, not in the identifier tag element. The presence of the identifier tag element results in the removal of the specified object, not in the removal of the entire hierarchy level represented by the container tag element.

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 "Deleting Configuration Elements" on page 103. For more information about identifiers, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16.

The following example shows how to remove the user object **barbara** from the **[edit system login user]** hierarchy level in the candidate configuration:

Client Application

```
<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>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2124

Deleting a Single-Value or Fixed-Form Option from a Configuration Object

To delete from a configuration object either a fixed-form option or an option that takes just one value, a client application includes the **operation="delete"** attribute in the tag element for the option. In the following, 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 Multivalue Option of a Configuration Object" on page 106.)

```
<configuration>
  <!-- opening tags for each parent of the object -->
```

```
      <object>
        <name>identifier</name> <!-- if object has an identifier -->
        <option1 operation="delete">
        <option2 operation="delete">
        <!-- 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 "Deleting Configuration Elements" on page 103. For more information about options, see "Mapping for Single-Value and Fixed-Form Leaf Statements" on page 19.

The following example shows how to remove the fixed-form **disable** option at the **[edit forwarding-options sampling]** hierarchy level:

Client Application

```
<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>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2125

Deleting Values from a Multivalue Option of a Configuration Object

As described in "Mapping for Leaf Statements with Multiple Values" on page 20, 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, 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 in the opening tag for each value. It does not include tag elements that represent values to be retained. The identifier tag element in the following is called **<name>**:

```
<configuration>
  <!-- opening tags for each parent of the parent object -->
    <parent-object>
      <name>identifier</name>
      <object operation="delete">value1</object>
      <object operation="delete">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 "Deleting Configuration Elements" on page 103. For more information about leaf statements with multiple values, see "Mapping for Leaf Statements with Multiple Values" on page 20.

The following example shows how to remove two of the permissions granted to the **user-accounts** login class:

Client Application

```
<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>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2126

CHAPTER 6

Committing Configurations

This chapter explains how to commit a candidate configuration so that 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 *Junos OS CLI User Guide*

Verifying a Configuration Before Committing It

During the process of committing the candidate configuration or a private copy, the NETCONF server confirms that it 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 candidate's correctness before actually committing it. A client application includes the **<validate>** and **<source>** tag elements and **<candidate/>** tag in an **<rpc>** tag element:

```
<rpc>
  <validate>
    <source>
      <candidate/>
    </source>
  </validate>
</rpc>
]]>]]>
```

The NETCONF server confirms that the candidate configuration 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 is not valid, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag element explaining the reason for the failure.

Committing a Configuration

The following sections describe how to commit the candidate configuration so that 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 commit operations, see the *Junos OS CLI User Guide*.

Committing the Candidate Configuration

To commit the candidate configuration, a client application encloses the **<commit/>** tag in an **<rpc>** tag element:

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

The NETCONF server confirms that it committed the candidate configuration by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot commit the candidate configuration, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag element explaining the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

To avoid inadvertently committing changes made by other users or applications, a client application locks the candidate configuration before changing it and emits the **<commit/>** tag while the configuration is still locked. (For instructions on locking and changing the candidate configuration, see "Locking the Candidate Configuration" on page 50 and "Changing Configuration Information" on page 85.) After committing the configuration, the client application unlocks the candidate as described in "Unlocking the Candidate Configuration" on page 51.

Committing the Candidate Configuration Only After Confirmation

To commit the candidate configuration but require an explicit confirmation for the commit to become permanent, a client application includes the **<confirmed/>** tag in **<commit>** and **<rpc>** tag elements:

```
<rpc>
  <commit>
    <confirmed/>
  </commit>
</rpc>
]]>]]>
```

If the commit is not confirmed within a certain amount of time (600 seconds [10 minutes] by default), the NETCONF server automatically retrieves and commits (rolls back to) the previously committed configuration. To specify a different number of minutes for the rollback deadline, the application encloses a positive integer value in the **<confirm-timeout>** tag element:

```
<rpc>
  <commit>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
  </commit>
</rpc>
]]>]]>
```

In either case, the NETCONF server confirms that it committed the candidate configuration temporarily by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

If the NETCONF server cannot commit the candidate, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag element explaining the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

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.

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>** tag element to specify how long to delay the next rollback; omit that tag element to delay the rollback by the default of 10 minutes. The client application can delay the rollback indefinitely by emitting the **<confirmed/>** tag repeatedly in this way.

To cancel the rollback completely (and commit a configuration permanently), the client application emits the **<commit/>** tag renclosed in an **<rpc>** tag element before the rollback deadline passes. The rollback is canceled and the candidate configuration is committed immediately, as described in "Committing a Configuration" on page 110. 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. For more information about session termination, see "Terminating Another NETCONF Session" on page 52.

The following example shows how to commit the candidate configuration with a rollback deadline of 20 minutes.

Client Application

```
<rpc>
  <commit>
    <confirmed/>
    <confirm-timeout>20</confirm-timeout>
  </commit>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

T2127

CHAPTER 7

Summary of NETCONF Tag Elements

This chapter lists the tag elements that client applications and the NETCONF server use to control the NETCONF session and to exchange configuration information. It also describes the **]]>]]>** character sequence, which signals the end of each request and response. The entries are in alphabetical order. For information about the notational conventions used in this chapter, see Table 2 on page xvi.

]]>]]>

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. Client applications send 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 .

Usage Guidelines

See "Generating Well-Formed XML Documents" on page 28.

Related Documentation

- <hello> on page 121
- <rpc> on page 124
- <rpc-reply> on page 125

<close-session/>

Usage

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

Description Request that the NETCONF server end the current session.

Usage Guidelines See "Ending a NETCONF Session and Closing the Connection" on page 53.

Related Documentation

-]]>]]> on page 113
- <rpc> on page 124

<commit>

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:

- To commit the configuration immediately, making it the active configuration on the device, emit the empty **<commit/>** tag.
- To commit the candidate configuration but require an explicit confirmation for the commit to become permanent, enclose the **<confirmed/>** tag in the **<commit>** tag element.

  By default, the NETCONF server rolls back to the previous running configuration after 10 minutes; to set a different rollback delay, also emit the optional **<confirm-timeout>** tag element. To delay the rollback again (past the original rollback deadline), emit the **<confirmed/>** tag (enclosed in the **<commit>** tag element) again before the deadline passes. Include the **<confirm-timeout>** tag element to specify how long to delay the next rollback, or omit that tag element to use the default of 10 minutes. The rollback can be delayed repeatedly in this way.

  To commit the configuration immediately and permanently after emitting the **<confirmed/>** tag, emit the empty **<commit/>** tag before the rollback deadline passes. The NETCONF server commits the candidate configuration and cancels the rollback.

If the candidate configuration is still the same as the running configuration, the effect is the same as recommitting the current running configuration.

Contents **<confirmed>**—Requests a temporary commit of the candidate configuration. The device reverts to the previous active configuration after a specified time.

<confirm-timeout>—Specifies the number of minutes before the device reverts to the previously active configuration. If this tag element is omitted, the default is 10 minutes.

Usage Guidelines See "Committing a Configuration" on page 110.

Related Documentation
-]]>]]> on page 113
- <rpc> on page 124

<copy-config>

Usage

```
<rpc>
  <copy-config>
    <target>
      <candidate/>
    </target>
    <source>
      <url>
        <!-- location specifier for file containing the new configuration -->
      </url>
    </source>
  <copy-config>
</rpc>
]]>]]>
```

Description Replace the existing candidate configuration with configuration data contained in a file.

Contents **<source>**—Encloses the **<url>** tag element, which specifies the source of the configuration data.

<url>—Names the file that contains the new configuration data to substitute for the existing candidate configuration. For information about specifying the file location, see "Referencing Configuration Data Files" on page 87.

The **<target>** tag element and its contents are explained separately.

Usage Guidelines See "Using <copy-config>" on page 96.

Related Documentation
-]]>]]> on page 113
- <rpc> on page 124
- <target> on page 126

<data>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration>
      <!-- Junos XML tag elements for the configuration data -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

Description Enclose configuration data returned by the NETCONF server in response to a **<get-config>** tag element.

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 configuration elements, see the *Junos XML API Configuration Reference*.

Usage Guidelines See "Requesting Configuration Information" on page 61.

Related Documentation

-]]>]]> on page 113
- **<configuration>** in the *Junos XML API Configuration Reference*
- <get-config> on page 120
- <rpc-reply> on page 125

<delete-config>

Usage

```
<rpc>
  <delete-config>
    <target>
      <candidate/>
    </target>
  <delete-config>
</rpc>
]]>]]>
```

Description Delete the existing candidate configuration.

Contents The **<target>** tag element and its contents are explained separately.

Usage Guidelines See "Replacing the Candidate Configuration" on page 95.

Related Documentation

-]]>]]> on page 113
- <rpc> on page 124

- <target> on page 126

<discard-changes/>

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.

Usage Guidelines See "Rolling Back a Configuration" on page 97.

Related Documentation

-]]>]]> on page 113
- <rpc> on page 124

<edit-config>

Usage

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>

    <!-- EITHER -->

    <config>
      <configuration>
        <!-- tag elements representing the data to incorporate -->
      </configuration>
    </config>

    <!-- OR -->

    <config-text>
      <configuration-text>
        <!-- tag elements inline configuration data in text format -->
      </configuration-text>
    </config-text>

    <!-- OR -->

    <url>
      <!-- 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. Provide the data in one of three ways:

- 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.
- Include the **<config-text>** tag element to provide a data stream of CLI configuration statemetnts to incorporate. The configuration statements are enclosed in the **<configuration-text>** tag element.
- Include the **<url>** tag element to specify the location of a file that contains the Junos XML configuration tag elements to incorporate.

Contents **<config>**—Encloses the **<configuration>** tag element.

<configuration>—Encloses the configuration data written in Junos XML. This configuration data will be incorporated into the candidate configuration and provided as a data stream. For information about the syntax for representing the elements to create, delete, or modify, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 16 and "Changing Individual Configuration Elements" on page 98.

<config-text>—Encloses the **<configuration-text>** tag element.

<configuration-text>—Encloses the configuration data written in CLI configuration statements. This configuration data will be incorporated into the candidate configuration and provided as a data stream. For information about the CLI configuration statements, see the *Junos OS CLI User Guide*.

<default-operation>—(Optional) Specifies how to incorporate the new configuration data into the candidate configuration, particularly when there are conflicting statements. The following are acceptable values:

- **merge**—Combines the new configuration data with the candidate configuration according to the rules defined in "Setting the Edit Configuration Mode" on page 91. This is the default mode if the **<default-operation>** tag element is omitted. It 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 (for information about the **operation** attribute, see "Changing Individual Configuration Elements" on page 98).

- **none**—Retains each configuration element in the existing candidate 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. For more information, see "Deleting Configuration Elements" on page 103.

- **replace**—Discards the existing candidate configuration and replaces it with the new data. For more information, see "Using <edit-config>" on page 96.

<error-option>—(Optional) Specifies how the NETCONF server handles errors encountered while it incorporates the configuration data. The following are acceptable values:

- **ignore-error**—Specifies that the NETCONF server continue to incorporate the new configuration data even if it encounters an error.

- **stop-on-error**—Specifies that the NETCONF server stop incorporating the new configuration data when it encounters an error. This is the default behavior if the **<error-option>** tag element is omitted.

<test-option>—(Optional) Specifies whether the NETCONF server validates the configuration data before incorporating it into the candidate configuration. The acceptable values defined in the NETCONF specification are **set** (no validation) and the default **test-then-set** (do not incorporate data if validation fails).

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. It performs a complete syntactic and semantic validation in response to the **<validate>** and **<commit>** tag elements, but not for the **<edit-config>** tag element.

<url>—Specifies the full pathname of the file that contains the configuration data to load. The file must reside on the routing platform's local disk. For more information, see "Referencing Configuration Data Files" on page 87.

The **<target>** tag element and its contents are explained separately.

Usage Guidelines

See "Changing Configuration Information" on page 85.

Related Documentation

-]]>]]> on page 113
- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc> on page 124
- <target> on page 126

<error-info>

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 Provide 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.

Usage Guidelines See "Handling an Error or Warning" on page 49.

Related Documentation

-]]>]]> on page 113
- <rpc-error> on page 124
- <rpc-reply> on page 125

<get-config>

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 the entire candidate configuration, enclose the **<source>** tag element and **<candidate/>** tag in the **<get-config>** tag element.
- 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/>—Represents the candidate configuration.

<configuration>—Encloses tag elements that specify which configuration elements to return.

<filter>—Encloses 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 syntax for representing each kind of element, see "Specifying the Scope of Configuration Information to Return" on page 65. For information about the configuration elements available in the current version of the Junos OS, see the *Junos XML API Configuration Reference*.

<running/>—Represents the active (mostly recently committed) configuration.

<source>—Encloses the tag that specifies the source of the configuration data. To specify the candidate configuration, include the **<candidate/>** tag. To specify the active configuration, include the **<running/>** tag.

Usage Guidelines

See "Requesting Configuration Information" on page 61.

Related Documentation

-]]>]]> on page 113
- **<configuration>** in the *Junos XML API Configuration Reference*
- <data> on page 116
- <rpc> on page 124

<hello>

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. For a list of the capabilities supported by the NETCONF server for the Junos OS, see "Exchanging <hello> Tag Elements" on page 40.

<session-id>—(Generated by NETCONF server only) Specifies the UNIX process ID (PID) of the NETCONF server for the session.

Usage Guidelines See "Exchanging <hello> Tag Elements" on page 40.

Related Documentation
-]]>]]> on page 113

<kill-session>

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 **maintenance** permission to perform this operation.

Contents <session-id>—The 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.

Usage Guidelines See "Terminating Another NETCONF Session" on page 52.

Related Documentation
-]]>]]> on page 113
- <lock> on page 123
- <rpc> on page 124

<lock>

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.

Usage Guidelines See "Locking the Candidate Configuration" on page 50.

Related Documentation
-]]>]]> on page 113
- <rpc> on page 124
- <target> on page 126
- <unlock> on page 126

<ok/>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

Description Indicate that the NETCONF server successfully performed a requested operation that changes the state or contents of the device configuration.

Usage Guidelines See "Configuration Change Responses" on page 48.

Related Documentation
-]]>]]> on page 113
- <rpc-reply> on page 125

<rpc>

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.

Usage Guidelines See "Sending a Request to the NETCONF Server" on page 43.

Related Documentation
-]]>]]> on page 113
- <rpc-reply> on page 125

<rpc-error>

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.

Contents **<error-message>**—Describes the error or warning in a natural-language text string.

<error-path>— Specifies the path to the Junos 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.

Usage Guidelines See "Handling an Error or Warning" on page 49.

Related Documentation

-]]>]]> on page 113
- <error-info> on page 120
- <rpc-reply> on page 125

<rpc-reply>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <!-- tag elements in a reply from the NETCONF server-->
</rpc-reply>
]]>]]>
```

Description Enclose all tag elements in a reply from the NETCONF server. The immediate child tag element is usually one of the following:

- The Junos XML tag element that encloses the data requested by a client application with a Junos XML operational request tag element; for example, the **<interface-information>** tag element in response to the **<get-interface-information>** tag element
- The **<data>** tag element, to enclose the data requested by a client application with the **<get-config>** tag element
- The **<ok/>** tag, to confirm that the NETCONF server successfully performed an operation that changes the state or contents of a configuration (such as a lock, change, or commit operation)
- The **<output>** tag element, if the Junos XML API does not define a specific tag element for requested operational information
- The **<rpc-error>** tag element, if the requested operation generated an error or warning

Attributes **xmlns**—Names the default XML namespace for the enclosed tag elements.

Usage Guidelines See "Parsing the NETCONF Server Response" on page 46.

Related Documentation

-]]>]]> on page 113
- <data> on page 116
- <ok/> on page 123
- <output> in the *Junos XML API Operational Reference*
- <rpc> on page 124
- <rpc-error> on page 124

<target>

Usage

```
<rpc>
  <( copy-config | delete-config | edit-config | lock | unlock )>
    <target>
      <candidate/>
    </target>
  </( copy-config | delete-config | edit-config | lock | unlock )>
</rpc>
]]>]]>
```

Description Specify the configuration on which to perform an operation.

Contents **<candidate/>**—Specifies the candidate configuration as the configuration on which to perform the operation. This is the only acceptable value for the Junos OS.

Usage Guidelines See "Locking the Candidate Configuration" on page 50, "Unlocking the Candidate Configuration" on page 51, "Editing the Candidate Configuration" on page 86, and "Using <copy-config>" on page 96.

Related Documentation

-]]>]]> on page 113
- <copy-config> on page 115
- <delete-config> on page 116
- <edit-config> on page 117
- <lock> on page 123
- <rpc> on page 124
- <unlock> on page 126

<unlock>

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.

Usage Guidelines See "Unlocking the Candidate Configuration" on page 51.

Related Documentation

-]]>]]> on page 113
- <lock> on page 123
- <rpc> on page 124
- <target> on page 126

<validate>

Usage

```
<rpc>
  <validate>
    <source>
      <candidate/>
    </source>
  </validate>
</rpc>
]]>]]>
```

Description Check that the candidate configuration is syntactically valid.

Contents **<source>**—Encloses the tag that specifies the configuration to validate.

<candidate/>—Represents the candidate configuration.

Usage Guidelines See "Verifying a Configuration Before Committing It" on page 109.

Related Documentation

-]]>]]> on page 113
- <rpc> on page 124

CHAPTER 8

Summary of Attributes in Junos XML Tags

This chapter describes the attributes that the NETCONF server and client applications include in opening Junos XML tags. For information about the notational conventions used in this chapter, see Table 2 on page xvi.

junos:changed-localtime

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) Specify the time when the configuration was last changed as the date and time in the device's local time zone.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:changed-seconds on page 129
- xmlns on page 132

junos:changed-seconds

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) Specify the time when the configuration was last changed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:changed-localtime on page 129
- xmlns on page 132

junos:commit-localtime

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) Specify the time when the configuration was committed as the date and time in the device's local time zone.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:commit-user on page 131
- junos:commit-seconds on page 130
- xmlns on page 132

junos:commit-seconds

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) Specify the time when the configuration was committed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:commit-user on page 131
- junos:commit-localtime on page 130
- xmlns on page 132

junos:commit-user

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) Specify the Junos username of the user who requested the commit operation.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:commit-localtime on page 130
- junos:commit-seconds on page 130
- xmlns on page 132

operation

Usage

```
<rpc>
  <edit-config>
    <config>
      <configuration>
        <!-- opening tags for each parent of the changing element -->
          <changing-element operation="( create | delete | replace )">
            <name>identifier</name>
            <!-- if changing element has an identifier -  ->
            <!-- other child tag elements, if appropriate for the operation -->
          </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 candidate configuration. If the attribute is omitted, the element is merged into the configuration according to the rules defined in "Setting the Edit Configuration Mode" on page 91. The following are acceptable values:

- **create**—Creates the specified element in the configuration only if the element does not already exist. See "Creating New Configuration Elements" on page 101.
- **delete**—Deletes the specified element from the candidate configuration. We recommend that the **<default-operation>** tag element with the value **none** also be included in the **<edit-config>** tag element. See "Deleting Configuration Elements" on page 103.
- **replace**—Replaces the specified element in the candidate configuration with the provided new configuration data. See "Replacing Configuration Elements" on page 100.

Usage Guidelines See "Changing Individual Configuration Elements" on page 98.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <edit-config> on page 117
- <rpc> on page 124
- xmlns on page 132

xmlns

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, define 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 "Requesting Operational Information" on page 60 and "Requesting Information from the Committed or Candidate Configuration" on page 63.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 125
- junos:changed-localtime on page 129
- junos:changed-seconds on page 129
- junos:commit-user on page 131
- junos:commit-localtime on page 130
- junos:commit-seconds on page 130

PART 3

Writing NETCONF Client Applications

CHAPTER 9

Writing NETCONF Perl Client Applications

Juniper Networks provides a Perl module, called **NET::Netconf::Manager**, to help you more quickly and easily develop custom NETCONF scripts for configuring and monitoring switches, routers, and security devices. The module implements a **NET::Netconf::Manager** object that client applications can use to communicate with the NETCONF server on a device. The Perl distribution includes several sample Perl scripts, which illustrate how to use the module in scripts that perform various functions.

This chapter discusses the following topics:

Overview of the NETCONF::Netconf::Manager Perl Module and Sample Scripts

The NETCONF Perl distribution uses the same directory structure for Perl modules as the Comprehensive Perl Archive Network (http://www.cpan.org/). This includes a **lib** directory for the **NET::Netconf** module and its supporting files, and an **examples** directory for the sample scripts.

Client applications use the **NET::Netconf::Manager** object to communicate with a NETCONF server. All of the sample scripts use this object.

The sample scripts illustrate how to perform the following functions:

- **diagnose_bgp.pl**—Illustrates how to write scripts to monitor device status and diagnose problems. The sample script extracts and displays information about a device's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data. The script is provided in the **examples/diagnose_bgp** directory in the NETCONF XML protocol Perl distribution.
- **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 to request the same information as the Junos XML <get-chassis-inventory><detail></get-chassis-inventory> tag sequence and the command-line interface (CLI) **show chassis hardware detail** command. The script is

provided in the **examples/get_chassis_inventory** directory in the NETCONF XML protocol Perl distribution.

- **edit_configuration.pl**—Illustrates how to change device configuration by loading a file that contains configuration data formatted with Junos XML tag elements. The distribution includes a sample configuration file, **config.xml**; however, you can specify another configuration file on the command line. The script is provided in the **examples/edit_configuration** directory in the NETCONF XML protocol Perl distribution.

For instructions on running the scripts, see the **README** or **README.html** file included in the NETCONF Perl distribution.

Download the NET::Netconf Module and Sample Scripts

To download the compressed tar archives that contains the NETCONF Perl client distribution and the prerequisites package, perform the following steps:

1. Access the Juniper Networks Customer Support Center Web page at http://www.juniper.net/customers/support/ .
2. Under Support, click the link labeled Download Software.
3. Under the Network Management section, click the link labeled NETCONF XML Management Protocol.
4. Click the link for the appropriate software release.
5. Select the Software tab.
6. Click the links labeled NETCONF API Perl client and NETCONF API Perl client prerequisites to download the client distribution and the prerequisites package.

NOTE: The NETCONF XML protocol Perl client software should be installed and run on a regular computer with a UNIX-like operating system; it is not meant to be installed on a Juniper Networks device.

7. Change to the directory where you want to create a subdirectory that contains the **NET::Netconf** Perl module and sample scripts:

   % cd *perl-parent-directory*

8. Issue the following command to uncompress and unpack the package downloaded in Step 6:

   - On FreeBSD and Linux systems:

     % tar zxf netconf-perl-*release-type***.tar.gz**

   - On Solaris systems:

     % gzip -dc netconf-perl-*release-type***.tar.gz | tar xf**

where *release* is the release code (such as **6.1R1.1**) and *type* is **domestic** or **export**. The command creates a directory called **netconf-perl-*release-type*** and writes the contents of the tar file to it.

9. See the **netconf-perl-*release-type*/README** file for instructions on unpacking and installing the Perl prerequisite modules, creating a **Makefile**, and installing and testing the **NET::Netconf** module.

Optionally, download the packages containing the document type definitions (DTDs) and the XML Schema language representation of the Junos configuration hierarchy:

1. Access the download page at http://www.juniper.net/support/products/xmlapi/ .
2. Click the link for the appropriate software release.
3. Select the Software tab.
4. Click the links to download the desired packages.

Tutorial: Writing Perl Client Applications

This tutorial explains how to write a Perl client application that requests operational or configuration information from the NETCONF server or loads configuration information onto a device. The following sections use the sample scripts included in the NETCONF XML Protocol Perl distribution as examples:

Import Perl Modules and Declare Constants

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 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 **diagnose_bgp.pl** script includes the following statement to declare a constant for the access method:

```
use constant VALID_ACCESS_METHOD => 'ssh';
```

The **edit_configuration.pl** 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;
```

Connect to the NETCONF Server

The following sections explain how to use the **NET::Netconf::Manager** object to connect to the NETCONF server on a device:

Satisfy Protocol Prerequisites

The NETCONF server supports several access protocols. For each connection to the NETCONF server on a device, the application must specify the protocol it is using. Client Perl 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 (**set system services netconf ssh**).

Group Requests

Establishing a connection to the NETCONF server on a device 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:

- The access protocol to use when communicating with the NETCONF server (key name: **access**). Before the application runs, satisfy the SSH prerequisites.
- The name of the device to which to connect (key name: hostname). For best results, specify either a fully-qualified hostname or an IP address.
- The username under which to establish the connection to the NETCONF server and issue requests (key name: **login**). The username must already exist on the specified device and have the permission bits necessary for making the requests invoked by the application.
- The password for the username (key name: **password**).

The sample scripts 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 obtain the parameters from options entered on the command line by a user. 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.

Example: Collect Parameters Interactively

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 script records the options in a Perl hash called **%opt**, using the **getopts** function defined in the **Getopt::Std** Perl module to read the options from the command line. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

In the following example from the **get_chassis_inventory.pl** script, 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:x: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** 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'.
 -x <format>   The name of the XSL file to display the response.
          Default: xsl/chassis_inventory.xsl
 -f <xmlfile>  The name of the XML file to print server response to.
          Default: xsl/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";
}
```

Obtain 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, or the name of the Extensible Stylesheet Transformation Language (XSLT) file to use for transforming the data.

As with the parameters required by the **NET::Netconf::Manager** object, your 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'} || "";
```

The following code from the **diagnose_bgp.pl** script defines which XSLT file to use to transform the NETCONF server's response. The first line sets the **$xslfile** variable to the value specified by the **-x** command-line option. If the option is not provided, the script uses the **text.xsl** file supplied with the script, which transforms the data to ASCII text. The **if** statement verifies that the specified XSLT file exists; the script terminates if it does not.

```
# Get the xsl file
my $xslfile = $opt{'x'} || "xsl/bgp.xsl";

# Check for the existence of the given file
if (! -f $xslfile) {
  croak "XSL file $xslfile does not exist.";
}
```

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";
}
```

Submitting a Request to the NETCONF Server

After establishing a connection to a NETCONF server (see "Submitting a Request to the NETCONF Server" on page 145), your application can submit one or more requests by invoking the Perl methods that are supported in the version of the NETCONF XML protocol and Junos XML API used by the application:

- Each version of software supports a set of methods that correspond to CLI operational mode commands (later releases generally support more methods). For a list of the operational methods supported in the current version, see the files stored in the **lib\Net\Netconf\Plugins\Plugin*release*** directory of the NETCONF Perl distribution (*release* is the Junos OS version code, such as **6.1R1** for the initial version of Junos OS Release 6.1). The files have names in the format ***package*_methods.pl**, where *package* is a software package.
- The set of methods that correspond to operations on configuration objects is defined in the **lib/Net/Netconf/Plugins.pm** file in the NETCONF distribution.

See the following sections for more information:

Providing Method Options or Attributes

Many Perl methods have one or more options or attributes. The following list describes the notation used to define a method's options in the **lib/Net/Netconf/Plugins.pm** and **lib/Net/Netconf/*release/package*_methods.pl** files, and the notation that an application uses when invoking the method:

- 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, follow 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_software_information** method takes 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 it to the value **1** (one) as in the following example:

```
$jnx->get-ancp-neighbor-information(brief => 1);
```

- 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-passive-monitoring-usage-information>
## Returns: <passive-monitoring-usage-information>
## Command: "show passive-monitoring usage"
get_passive_monitoring_usage_information => {
  interface_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 an 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, attributes, 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 is the recommended way to send a request to the NETCONF server and shows how to handle error conditions. The **$jnx** variable is defined to be a **NET::Netconf::Manager** object.

```
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',
  '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);
```

Example: Get an Inventory of Hardware Components

The **get_chassis_inventory.pl** script retrieves and displays a detailed inventory of the hardware components installed in a routing, swtiching, or security platform. It is equivalent to issuing the **show chassis hardware detail** command.

After establishing a connection to the NETCONF server, the script defines **get_chassis_inventory** as the request to send and includes the **detail** argument:

```
my $query = "get_chassis_inventory";
my %queryargs = ( 'detail' => 1 );
```

The script sends the query and assigns the results to the **$res** variable. It performs two tests on the results, and prints an error message if it cannot send the request or if errors occurred when executing it. If no errors occurred, the script uses XSLT to transform the results.

```
# 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 {
  # Transform the server response using XSL file
  my $res = new Net::Netconf::Transform();
  print "Transforming ...\n";
  my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile,
                        "$xslfile.tmp",
                        $xmlfile);
  if ($nm) {
    format_by_xslt($nm, $xmlfile, );
  } else {
    print STDERR "ERROR: Invalid XSL File $xslfile\n";
  }
}

# Disconnect from the Netconf server
$jnx->disconnect();
```

Example: Edit Configuration Statements

The **edit_configuration.pl** script edits configuration statements and loads the configuration onto 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 in a slightly more elaborate manner than the generic structure described in "Handling Error Conditions" on page 149. 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. In case of error, the script invokes the **graceful_shutdown** subroutine described in "Handling Error Conditions" on page 149.

```
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 return its contents in 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 now invokes the **edit_config** method to edit the candidate configuration on 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, the script invokes the **commit** method:

```
# 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);
}
```

Parsing and Formatting the Response from the NETCONF Server

As the last step in sending a request, the application verifies that there are no errors with the response from the NETCONF server. It can then write the response to a file, to the screen, or both. If the response is for an operational query, the application usually uses XSLT to transform the output into a more readable format, such as HTML or formatted ASCII. If the response consists of configuration data, the application can store it as XML (the Junos XML tag elements generated by default from the NETCONF server) or transform it into formatted ASCII text.

The following sections discuss parsing and formatting options:

- Parsing and Formatting an Operational Response on page 152

Parsing and Formatting an Operational Response

The following code sample from the **diagnose_bgp.pl** script uses XSLT to transform an operational response from the NETCONF server into a more readable format. A detailed discussion of the functional subsections follows the complete code sample.

```
# Get the output file
my $outputfile = $opt{'o'} || "";
```

```
# Get the xsl file
my $xslfile = $opt{'x'} || "xsl/bgp.xsl";

# Check for the existence of the given file
if (! -f $xslfile) {
  croak "XSL file $xslfile does not exist.";
}

# Get the xmlfile
my $xmlfile = $opt{'f'} || "xsl/bgp.xml";

# send the command and get the server response
my $res = $jnx->$query();

# 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 {
  # Transform the server response using XSL file
  my $res = new Net::Netconf::Transform();
  print "Transforming ...\n";
  my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile,
                        "$xslfile.tmp",
                        $xmlfile);
  if ($nm) {
    format_by_xslt($nm, $xmlfile, );
  } else {
    print STDERR "ERROR: Invalid XSL File $xslfile\n";
  }
}
```

The first line of the preceding code sample illustrates how the scripts read the **-o** option from the command line to obtain the name of the file into which to write the results of the XSLT transformation:

```
my $outputfile = $opt{'o'} || "";
```

From the **-x** command-line option, the scripts obtain the name of the XSLT file to use, setting a default value if the option is not provided. The scripts exit if the specified file does not exist. The following example is from the **diagnose_bgp.pl** script:

```
my $xslfile = $opt{'x'} || "xsl/bgp.xsl";
if (! -f $xslfile) {
  croak "XSL file $xslfile does not exist.";
}
```

For examples of XSLT files, see the following directories in the NETCONF Perl distribution:

- The **examples/diagnose_bpg/xsl** directory contains an XSLT file for the **diagnose_bpg.pl** script.
- The **examples/get_chassis_inventory/xsl** directory contains XSLT files for the **get_chassis_inventory.pl** script.

The actual parsing operation invokes the **translateXSLtoRelease** function (defined in the **Net::Netconf::Transform** module) to alter one of the namespace definitions in the XSLT file.

```
my $res = new Net::Netconf::Transform();
  print "Transforming ...\n";
  my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile,
                        "$xslfile.tmp",
                        $xmlfile);
  if ($nm) {
    format_by_xslt($nm, $xmlfile, );
  } else {
    print STDERR "ERROR: Invalid XSL File $xslfile\n";
  }
```

This is necessary because the XSLT 1.0 specification requires that every XSLT file define a specific value for each default namespace used in the data being transformed. The **xmlns** attribute in a NETCONF operational response tag element includes a code representing the Junos OS version, such as**10.3R1** for the initial version of Junos OS Release 10.3. Because the same XSLT file can be applied to operational response tag elements from devices running different versions of the Junos OS, the XSLT file cannot predefine an **xmlns** namespace value that matches all versions. The **translateXSLtoRelease** function alters the namespace definition in the XSLT file identified by the **$xslfile** variable to match the value in the NETCONF server's response. It assigns the resulting XSLT file to the **$nm** variable.

After verifying that the **translateXSLtoRelease** function succeeded, the script invokes the **format_by_xslt** function, which builds a command string and assigns it to the **$command** variable. The first part of the command string invokes the **xsltproc** command and specifies the names of the XSLT and configuration data files (**$xslfile** and **$xmlfile**)::

```
sub format_by_xslt
{
  my ($xslfile, $xmlfile, $outfile)  = @_;

  print "Transforming $xmlfile with $xslfile...\n" if $outfile;
  my $command = "xsltproc $xslfile $xmlfile";
  $command .= "> $outfile" if $outfile;
  system($command);
  print "Done\n" if $outfile;
  print "See $outfile\n" if $outfile;
}
```

If the **$outfile** variable is defined (the file for storing the result of the XSLT transformation exists), the script appends a string to the **$command** variable to write the results of the **xsltproc** command to the file. (If the file does not exist, the script writes the results to standard out [stdout].) The script then invokes the **system** function to execute the command string and prints status messages to stdout.

If the **translateXSLtoRelease** function fails (the **if ($nm)** expression evaluates to “false”), the script prints an error:

```
if ($nm) {
   format_by_xslt($nm, $xmlfile, );
} else {
```

```
    print STDERR "ERROR: Invalid XSL File $xslfile\n";
}
```

Closing the Connection to the NETCONF Server

To end the NETCONF session and close the connection to the device, each sample script invokes the **disconnect** method. Several of the scripts do this in standalone statements:

```
$jnx->disconnect();
```

The **edit_configuration.pl** script invokes the **graceful_shutdown** method instead.

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

The **graceful_shutdown** method takes the appropriate actions with regard to the configuration database and then invokes the **disconnect** method.

PART 4

Index

Index

Symbols

A

B

C

D

E

F

G

S

T

U

V

W

X

Index of Statements and Commands