JUNIPER NETWORKS

JUNOS® OS

Junos XML Management Protocol Guide

Release

10.4

Published: 2010-10-19

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

JUNOS® OS Junos XML Management Protocol Guide
Release 10.4

Writing: Tony Mauro, 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 listed in the revision history.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. The 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

READ THIS END USER LICENSE AGREEMENT ("AGREEMENT") BEFORE DOWNLOADING, INSTALLING, OR USING THE SOFTWARE. BY DOWNLOADING, INSTALLING, OR USING THE SOFTWARE OR OTHERWISE EXPRESSING YOUR AGREEMENT TO THE TERMS CONTAINED HEREIN, YOU (AS CUSTOMER OR IF YOU ARE NOT THE CUSTOMER, AS A REPRESENTATIVE/AGENT AUTHORIZED TO BIND THE CUSTOMER) CONSENT TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT OR CANNOT AGREE TO THE TERMS CONTAINED HEREIN, THEN (A) DO NOT DOWNLOAD, INSTALL, OR USE THE SOFTWARE, AND (B) YOU MAY CONTACT JUNIPER NETWORKS REGARDING LICENSE TERMS.

1. **The Parties.** The parties to this Agreement are (i) Juniper Networks, Inc. (if the Customer's principal office is located in the Americas) or Juniper Networks (Cayman) Limited (if the Customer's principal office is located outside the Americas) (such applicable entity being referred to herein as "Juniper"), and (ii) the person or organization that originally purchased from Juniper or an authorized Juniper reseller the applicable license(s) for use of the Software ("Customer") (collectively, the "Parties").

2. **The Software.** In this Agreement, "Software" means the program modules and features of the Juniper or Juniper-supplied software, for which Customer has paid the applicable license or support fees to Juniper or an authorized Juniper reseller, or which was embedded by Juniper in equipment which Customer purchased from Juniper or an authorized Juniper reseller. "Software" also includes updates, upgrades and new releases of such software. "Embedded Software" means Software which Juniper has embedded in or loaded onto the Juniper equipment and any updates, upgrades, additions or replacements which are subsequently embedded in or loaded onto the equipment.

3. **License Grant.** Subject to payment of the applicable fees and the limitations and restrictions set forth herein, Juniper grants to Customer a non-exclusive and non-transferable license, without right to sublicense, to use the Software, in executable form only, subject to the following use restrictions:

a. Customer shall use Embedded Software solely as embedded in, and for execution on, Juniper equipment originally purchased by Customer from Juniper or an authorized Juniper reseller.

b. Customer shall use the Software on a single hardware chassis having a single processing unit, or as many chassis or processing units for which Customer has paid the applicable license fees; provided, however, with respect to the Steel-Belted Radius or Odyssey Access Client software only, Customer shall use such Software on a single computer containing a single physical random access memory space and containing any number of processors. Use of the Steel-Belted Radius or IMS AAA software on multiple computers or virtual machines (e.g., Solaris zones) requires multiple licenses, regardless of whether such computers or virtualizations are physically contained on a single chassis.

c. Product purchase documents, paper or electronic user documentation, and/or the particular licenses purchased by Customer may specify limits to Customer's use of the Software. Such limits may restrict use to a maximum number of seats, registered endpoints, concurrent users, sessions, calls, connections, subscribers, clusters, nodes, realms, devices, links, ports or transactions, or require the purchase of separate licenses to use particular features, functionalities, services, applications, operations, or capabilities, or provide throughput, performance, configuration, bandwidth, interface, processing, temporal, or geographical limits. In addition, such limits may restrict the use of the Software to managing certain kinds of networks or require the Software to be used only in conjunction with other specific Software. Customer's use of the Software shall be subject to all such limitations and purchase of all applicable licenses.

d. For any trial copy of the Software, Customer's right to use the Software expires 30 days after download, installation or use of the Software. Customer may operate the Software after the 30-day trial period only if Customer pays for a license to do so. Customer may not extend or create an additional trial period by re-installing the Software after the 30-day trial period.

e. The Global Enterprise Edition of the Steel-Belted Radius software may be used by Customer only to manage access to Customer's enterprise network. Specifically, service provider customers are expressly prohibited from using the Global Enterprise Edition of the Steel-Belted Radius software to support any commercial network access services.

The foregoing license is not transferable or assignable by Customer. No license is granted herein to any user who did not originally purchase the applicable license(s) for the Software from Juniper or an authorized Juniper reseller.

4. **Use Prohibitions.** Notwithstanding the foregoing, the license provided herein does not permit the Customer to, and Customer agrees not to and shall not: (a) modify, unbundle, reverse engineer, or create derivative works based on the Software; (b) make unauthorized copies of the Software (except as necessary for backup purposes); (c) rent, sell, transfer, or grant any rights in and to any copy of the Software, in any form, to any third party; (d) remove any proprietary notices, labels, or marks on or in any copy of the Software or any product in which the Software is embedded; (e) distribute any copy of the Software to any third party, including as may be embedded in Juniper equipment sold in the secondhand market; (f) use any 'locked' or key-restricted feature, function, service, application, operation, or capability without first purchasing the applicable license(s) and obtaining a valid key from Juniper, even if such feature, function, service, application, operation, or capability is enabled without a key; (g) distribute any key for the Software provided by Juniper to any third party; (h) use the

Software in any manner that extends or is broader than the uses purchased by Customer from Juniper or an authorized Juniper reseller; (i) use Embedded Software on non-Juniper equipment; (j) use Embedded Software (or make it available for use) on Juniper equipment that the Customer did not originally purchase from Juniper or an authorized Juniper reseller; (k) disclose the results of testing or benchmarking of the Software to any third party without the prior written consent of Juniper; or (l) use the Software in any manner other than as expressly provided herein.

5. **Audit.** Customer shall maintain accurate records as necessary to verify compliance with this Agreement. Upon request by Juniper, Customer shall furnish such records to Juniper and certify its compliance with this Agreement.

6. **Confidentiality.** The Parties agree that aspects of the Software and associated documentation are the confidential property of Juniper. As such, Customer shall exercise all reasonable commercial efforts to maintain the Software and associated documentation in confidence, which at a minimum includes restricting access to the Software to Customer employees and contractors having a need to use the Software for Customer's internal business purposes.

7. **Ownership.** Juniper and Juniper's licensors, respectively, retain ownership of all right, title, and interest (including copyright) in and to the Software, associated documentation, and all copies of the Software. Nothing in this Agreement constitutes a transfer or conveyance of any right, title, or interest in the Software or associated documentation, or a sale of the Software, associated documentation, or copies of the Software.

8. **Warranty, Limitation of Liability, Disclaimer of Warranty.** The warranty applicable to the Software shall be as set forth in the warranty statement that accompanies the Software (the "Warranty Statement"). Nothing in this Agreement shall give rise to any obligation to support the Software. Support services may be purchased separately. Any such support shall be governed by a separate, written support services agreement. TO THE MAXIMUM EXTENT PERMITTED BY LAW, JUNIPER SHALL NOT BE LIABLE FOR ANY LOST PROFITS, LOSS OF DATA, OR COSTS OR PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THIS AGREEMENT, THE SOFTWARE, OR ANY JUNIPER OR JUNIPER-SUPPLIED SOFTWARE. IN NO EVENT SHALL JUNIPER BE LIABLE FOR DAMAGES ARISING FROM UNAUTHORIZED OR IMPROPER USE OF ANY JUNIPER OR JUNIPER-SUPPLIED SOFTWARE. EXCEPT AS EXPRESSLY PROVIDED IN THE WARRANTY STATEMENT TO THE EXTENT PERMITTED BY LAW, JUNIPER DISCLAIMS ANY AND ALL WARRANTIES IN AND TO THE SOFTWARE (WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE), INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT DOES JUNIPER WARRANT THAT THE SOFTWARE, OR ANY EQUIPMENT OR NETWORK RUNNING THE SOFTWARE, WILL OPERATE WITHOUT ERROR OR INTERRUPTION, OR WILL BE FREE OF VULNERABILITY TO INTRUSION OR ATTACK. In no event shall Juniper's or its suppliers' or licensors' liability to Customer, whether in contract, tort (including negligence), breach of warranty, or otherwise, exceed the price paid by Customer for the Software that gave rise to the claim, or if the Software is embedded in another Juniper product, the price paid by Customer for such other product. Customer acknowledges and agrees that Juniper has set its prices and entered into this Agreement in reliance upon the disclaimers of warranty and the limitations of liability set forth herein, that the same reflect an allocation of risk between the Parties (including the risk that a contract remedy may fail of its essential purpose and cause consequential loss), and that the same form an essential basis of the bargain between the Parties.

9. **Termination.** Any breach of this Agreement or failure by Customer to pay any applicable fees due shall result in automatic termination of the license granted herein. Upon such termination, Customer shall destroy or return to Juniper all copies of the Software and related documentation in Customer's possession or control.

10. **Taxes.** All license fees payable under this agreement are exclusive of tax. Customer shall be responsible for paying Taxes arising from the purchase of the license, or importation or use of the Software. If applicable, valid exemption documentation for each taxing jurisdiction shall be provided to Juniper prior to invoicing, and Customer shall promptly notify Juniper if their exemption is revoked or modified. All payments made by Customer shall be net of any applicable withholding tax. Customer will provide reasonable assistance to Juniper in connection with such withholding taxes by promptly: providing Juniper with valid tax receipts and other required documentation showing Customer's payment of any withholding taxes; completing appropriate applications that would reduce the amount of withholding tax to be paid; and notifying and assisting Juniper in any audit or tax proceeding related to transactions hereunder. Customer shall comply with all applicable tax laws and regulations, and Customer will promptly pay or reimburse Juniper for all costs and damages related to any liability incurred by Juniper as a result of Customer's non-compliance or delay with its responsibilities herein. Customer's obligations under this Section shall survive termination or expiration of this Agreement.

11. **Export.** Customer agrees to comply with all applicable export laws and restrictions and regulations of any United States and any applicable foreign agency or authority, and not to export or re-export the Software or any direct product thereof in violation of any such restrictions, laws or regulations, or without all necessary approvals. Customer shall be liable for any such violations. The version of the Software supplied to Customer may contain encryption or other capabilities restricting Customer's ability to export the Software without an export license.

12. **Commercial Computer Software.** The Software is "commercial computer software" and is provided with restricted rights. Use, duplication, or disclosure by the United States government is subject to restrictions set forth in this Agreement and as provided in DFARS 227.7201 through 227.7202-4, FAR 12.212, FAR 27.405(b)(2), FAR 52.227-19, or FAR 52.227-14(ALT III) as applicable.

13. **Interface Information.** To the extent required by applicable law, and at Customer's written request, Juniper shall provide Customer with the interface information needed to achieve interoperability between the Software and another independently created program, on payment of applicable fee, if any. Customer shall observe strict obligations of confidentiality with respect to such information and shall use such information in compliance with any applicable terms and conditions upon which Juniper makes such information available.

14. **Third Party Software.** Any licensor of Juniper whose software is embedded in the Software and any supplier of Juniper whose products or technology are embedded in (or services are accessed by) the Software shall be a third party beneficiary with respect to this Agreement, and such licensor or vendor shall have the right to enforce this Agreement in its own name as if it were Juniper. In addition, certain third party software may be provided with the Software and is subject to the accompanying license(s), if any, of its respective owner(s). To the extent portions of the Software are distributed under and subject to open source licenses obligating Juniper to make the source code for such portions publicly available (such as the GNU General Public License ("GPL") or the GNU Library General Public License ("LGPL")), Juniper will make such source code portions (including Juniper modifications, as appropriate) available upon request for a period of up to three years from the date of distribution. Such request can be made in writing to Juniper Networks, Inc., 1194 N. Mathilda Ave., Sunnyvale, CA 94089, ATTN: General Counsel. You may obtain a copy of the GPL at http://www.gnu.org/licenses/gpl.html, and a copy of the LGPL at http://www.gnu.org/licenses/lgpl.html .

15. **Miscellaneous.** This Agreement shall be governed by the laws of the State of California without reference to its conflicts of laws principles. The provisions of the U.N. Convention for the International Sale of Goods shall not apply to this Agreement. For any disputes arising under this Agreement, the Parties hereby consent to the personal and exclusive jurisdiction of, and venue in, the state and federal courts within Santa Clara County, California. This Agreement constitutes the entire and sole agreement between Juniper and the Customer with respect to the Software, and supersedes all prior and contemporaneous agreements relating to the Software, whether oral or written (including any inconsistent terms contained in a purchase order), except that the terms of a separate written agreement executed by an authorized Juniper representative and Customer shall govern to the extent such terms are inconsistent or conflict with terms contained herein. No modification to this Agreement nor any waiver of any rights hereunder shall be effective unless expressly assented to in writing by the party to be charged. If any portion of this Agreement is held invalid, the Parties agree that such invalidity shall not affect the validity of the remainder of this Agreement. This Agreement and associated documentation has been written in the English language, and the Parties agree that the English version will govern. (For Canada: Les parties aux présentés confirment leur volonté que cette convention de même que tous les documents y compris tout avis qui s'y rattaché, soient redigés en langue anglaise. (Translation: The parties confirm that this Agreement and all related documentation is and will be in the English language)).

Abbreviated Table of Contents

Table of Contents

List of Tables

About This Guide

This preface provides the following guidelines for using the *JUNOS® OS Junos 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 Junos Extensible Markup Language (XML) management protocol and the Junos XML application programming interface (API) to

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

NOTE: For additional information about 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) device 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 tags and attributes.

Documentation Conventions

Table 1 on page xxi 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 xxii 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 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; IP addresses; 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 Junos 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 on page 5
- XML Overview on page 6
- *Junos XML API Configuration Reference*

- *Junos XML API Operational Reference*

Junos XML API and Junos XML Management Protocol Overview

The Junos XML Management Protocol is an XML-based protocol that client applications use to request and change configuration information on routing, switching, and security platforms running Junos OS. It uses an XML-based data encoding for the configuration data and remote procedure calls. The 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 Junos 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 device.

Client applications request or change information on a device by encoding the request with tag elements from the Junos XML management protocol and Junos XML API and sending it to the Junos XML protocol server on the device. The Junos XML protocol server is integrated into the Junos OS and does not appear as a separate entry in process listings. The Junos XML protocol server directs the request to the appropriate software modules within the device, encodes the response in Junos XML and Junos XML protocol 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 Junos XML API **<get-interface-information>** request tag element . The Junos XML protocol server gathers the information from the interface process and returns it in the Junos XML API **<interface-information>** response tag element.

You can use the Junos 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 Junos XML protocol server, and you can also utilize the Junos XML protocol to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

Related Documentation

- Advantages of Using the Junos XML Management Protocol and Junos XML API on page 8
- XML and the Junos OS on page 3
- XML Overview on page 6

XML Overview

Extensible Markup Language (XML) is a language for defining a set of markers, called *tags*, that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. 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 general aspects of XML:

Tag Elements

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

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

The term *tag element* 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 the tag element. 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 the documentation to indicate optional parts of Junos OS CLI command strings.

Junos XML and Junos XML protocol tag elements obey the XML convention that the tag element name indicates the kind of information enclosed by the tags. 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 device, 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 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.

Attributes

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

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

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

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

Namespaces

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

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

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

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

During processing, the prefix is expanded into the URI reference. Although there may be multiple namespaces that define an **output** element or function, the use of **jcs:output**

explicitly defines which **output** function is used. You can choose any prefix to refer to the contents in a namespace, but there must be an existing declaration in the XML document that binds the prefix to the associated URI.

Document Type Definition

An XML-tagged document or data set is *structured*, because a set of rules specifies the ordering and interrelationships of the items in it. 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.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 5
- XML and the Junos OS on page 3

Advantages of Using the Junos XML Management Protocol and Junos XML API

The Junos XML management protocol and Junos XML API fully document all options for every supported Junos operational request, all statements in the Junos configuration hierarchy, and basic operations that are equivalent to configuration mode commands. The tag names clearly indicate the function of an element in an operational or configuration request or a 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. Junos XML and Junos XML protocol tag elements make it straightforward for client applications that request information from a device to parse the output and find specific information.

Parsing Device Output

The following example illustrates how the Junos XML API makes it easier to parse device output and extract the needed information. The example compares formatted ASCII and XML-tagged versions of output from a device running 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 **SNMP ifIndex** and also account for the included comma.

A problem arises if the format or ordering of text output changes in a later version of the Junos OS. For example, if a **Logical index** field is added following the interface index number, the new formatted ASCII might appear as follows:

```
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 **Logical index:** label as the new delimiter.

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 Junos XML protocol 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.

Displaying Device Output

XML-tagged output is also easier to transform into different display formats than formatted ASCII output. 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 write special routines and data structures in your display program to extract and show the appropriate information 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.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 5
- XML Overview on page 6

Overview of a Junos XML Protocol Session

Communication between the Junos XML protocol 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 Junos XML protocol server constitutes a *well-formed* XML document, because the tag streams obey the structural rules defined in the Junos XML protocol 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 Junos XML protocol session.

1. The client application establishes a connection to the Junos XML protocol server and opens the Junos XML protocol session.
2. The Junos XML protocol server and client application exchange initialization information, which is used to determine if they are using compatible versions of the Junos OS and the Junos XML management protocol.
3. The client application sends one or more requests to the Junos XML protocol server and parses its responses.
4. The client application closes the Junos XML protocol session and the connection to the Junos XML protocol server.

CHAPTER 2

Using Junos XML Management Protocol and Junos XML Tag Elements

This chapter describes the syntactic and notational conventions used by the Junos XML protocol 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 Junos XML Management Protocol Conventions Overview

A client application must comply with XML and Junos 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 Junos XML protocol and Junos XML document type definitions (DTDs) for the kind of information encoded in the request. The client application must emit tag elements in the required order and only in legal contexts. Compliant applications are easier to maintain in the event of changes to the Junos OS or Junos XML management protocol.

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

The following sections describe Junos 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 Junos XML Protocol Server" on page 48.

A *response* tag element represents the Junos XML protocol 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 Junos XML Protocol Server Response" on page 51.

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 Junos XML protocol server returns the **<interface-information>** response tag element.

Client Application

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

Junos XML Protocol Server

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

T1100

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 Junos XML protocol 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 Junos XML Protocol Server Response" on page 51. For information about the **xmlns** attribute in the opening **<interface-information>** tag, see "Requesting Operational Information" on page 66.

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>**.

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 Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol server preserves the white space in the configuration database.

In its responses, the Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol server gracefully but must not depend on their content. Client applications also cannot use comments to convey information to the Junos XML protocol 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. -->
```

XML Processing Instructions

An XML processing instruction (PI) contains information relevant to a particular protocol and has the following form:

```
<?PI-name attributes?>
```

Some PIs emitted during a Junos XML protocol session include information that a client application needs for correct operation. A prominent example is the **<?xml?>** tag element, which the client application and Junos XML protocol server each emit at the beginning of every Junos XML protocol session to specify which version of XML and which character encoding scheme they are using. For more information, see “Emitting the <?xml?> PI” on page 41 and “Parsing the Junos XML Protocol Server’s <?xml?> PI” on page 43.

The Junos XML protocol server can also emit PIs that the client application does not need to interpret (for example, PIs intended for the CLI). If the client application does not understand a PI, it must treat the PI like a comment instead of exiting or generating an error message.

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 Junos XML protocol 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 15 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 17

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	JUNOS XML Tags
system {	<configuration>
login {	<system>
...*child statements*...	<login>
}	<!- - *tags for child statements* - ->
}	</login>
protocols {	</system>
ospf {	<protocols>
...*child statements*...	<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 21.

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 111.

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 111.

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 Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol server's response to 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 Junos XML protocol server's response, it is certain to be syntactically correct. For more information about changing routing platform configuration, see "Changing Configuration Information" on page 111.

Similarly, when a client application requests information about a configuration element (hierarchy level or configuration object), it uses the same tag elements that the Junos XML protocol 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 Junos XML protocol 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 68.

The tag streams emitted by the Junos XML protocol 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 Junos XML Management Protocol

CHAPTER 3

Controlling the Junos XML Management Protocol Session

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

Client Application's Role in a Junos XML Protocol Session

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

1. Establishes a connection to the Junos XML protocol server on the routing platform, as described in "Connecting to the Junos XML Protocol Server" on page 39.
2. Opens a Junos XML protocol session, as described in "Starting the Junos XML Protocol Session" on page 40.
3. (Optional) Locks the candidate configuration or creates a private copy, as described in "Exchanging Information with the Junos XML Protocol Server" on page 48. Locking the configuration prevents other users or applications from changing it at the same time. Creating a private copy of the configuration enables the application to make changes without affecting the candidate or active configuration until the copy is committed.

4. Requests operational or configuration information, or changes configuration information, as described in "Requesting Information" on page 65 and "Changing Configuration Information" on page 111.
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 141.
6. Commits changes made to the configuration, as described in "Committing a Configuration" on page 141.
7. Unlocks the candidate configuration if it is locked, as described in "Unlocking the Candidate Configuration" on page 56.
8. Ends the Junos XML protocol session and closes the connection to the device, as described in "Ending a Junos XML Protocol Session and Closing the Connection" on page 58.

Establishing a Junos XML Management Protocol Session

The Junos XML protocol server communicates with client applications within the context of a Junos XML protocol *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 Junos XML protocol tag elements and Junos XML tag elements emitted by the Junos XML protocol 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 Junos XML protocol server using one of the protocols listed in "Supported Access Protocols" on page 29. To authenticate with the Junos XML protocol server, they use either a Junos XML protocol-specific mechanism or the access protocol's standard authentication mechanism, depending on the protocol. After authentication, the Junos XML protocol 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 a Junos XML protocol session, see the following sections:

- Supported Access Protocols on page 29
- Prerequisites for Establishing a Connection on page 29
- Connecting to the Junos XML Protocol Server on page 39
- Starting the Junos XML Protocol Session on page 40
- Authenticating with the Junos XML Protocol Server on page 45

For an example of a complete Junos XML protocol session, see "Example of a Junos XML Protocol Session" on page 60.

Supported Access Protocols

To connect to the Junos XML protocol server, client applications can use the access protocols and associated authentication mechanisms listed in Table 5 on page 29.

Table 5: Supported Access Protocols and Authentication Mechanisms

Access Protocol	Authentication Mechanism
clear-text, a Junos XML protocol-specific access protocol for sending unencrypted text over a Transmission Control Protocol (TCP) connection	Junos XML protocol-specific
SSH	Standard SSH
Outbound SSH	Outbound SSH
Secure Sockets Layer (SSL)	Junos XML protocol-specific
Telnet	Standard Telnet

The SSH and SSL protocols are preferred because they encrypt security information (such as passwords) before transmitting it across the network. Outbound SSH allows you to create an encrypted connection to the device in situations where you cannot connect to the device using standard SSH. The clear-text and Telnet protocols do not encrypt information.

For information about the prerequisites for each access protocol, see "Prerequisites for Establishing a Connection" on page 29. For authentication instructions, see "Authenticating with the Junos XML Protocol Server" on page 45.

Prerequisites for Establishing a Connection

To enable a client application to establish a connection to the Junos XML protocol server, you must satisfy the requirements discussed in the following sections:

Prerequisites for All Access Protocols

A client application must be able to log in to each device on which it establishes a connection with the Junos XML protocol server. The following instructions explain how to create a Junos login account for the application; for detailed information, see the chapter about configuring user access in the *Junos OS System Basics Configuration Guide*.

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 the 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:

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. You can also include the optional **full-name** and **uid** statements. Optionally, include the **full-name** and **uid** statements.

   ```
   [edit system login]
   user@host# set user account-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. Create a text-based password for the account by including either the **plain-text-password** or **encrypted-password** statement at the **[edit system login user *account-name* authentication]** hierarchy level.

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

   NOTE: A text-based password is not strictly necessary if the account is used to access the Junos XML protocol server through SSH with public/private key pairs for authentication, but we recommend that you create one anyway. The key pair alone is sufficient if the account is used only for SSH access, but a password is required if the account is also used for any other type of access (for login on the console, for example). The password is also used—the SSH server prompts for it—if key-based authentication is configured but fails. For information about creating a public/private key pair, see "Prerequisites for SSH Connections" on page 32.

   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"
```

3. Issue the **commit** command.

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

4. Repeat the preceding steps on each device where the client application establishes Junos XML protocol sessions.
5. Enable the client application to access the password and provide it when the Junos XML protocol server prompts for it. There are several possible methods, including the following:
   - Code the application to prompt the user for a password at startup and to store the password temporarily in a secure manner.
   - Store the password in encrypted form in a secure local-disk location or secured database and code the application to access it.

Prerequisites for Clear-Text Connections

A client application that uses the Junos XML protocol-specific clear-text access protocol sends unencrypted text directly over a TCP connection without using any additional protocol (such as SSH, SSL, or Telnet).

NOTE: Devices running the Junos-FIPS software do not accept Junos XML protocol clear-text connections. We recommend that you do not use the clear-text protocol in a Common Criteria environment. For more information, see the *Secure Configuration Guide for Common Criteria and Junos-FIPS*.

To enable client applications to use the clear-text protocol to connect to the Junos XML protocol server, perform the following steps:

1. Verify that the application can access the TCP software. On most operating systems, TCP is accessible in the standard distribution. Do this on each computer where the application runs.
2. Satisfy the prerequisites discussed in “Prerequisites for All Access Protocols” on page 29.
3. Configure the device running Junos OS to accept clear-text connections from client applications on port 3221 by including the **xnm-clear-text** statement at the **[edit system services]** hierarchy level:

```
[edit]
user@host# set system services xnm-clear-text
```

By default, the Junos XML protocol server supports up to 75 simultaneous clear-text sessions and 150 connection attempts per minute. Optionally, you can include either or both the **connection-limit** statement to limit the number of concurrent sessions and the **rate-limit** statement to limit the number of connection attempts. Both statements accept a value from **1** through **250**.

```
[edit]
user@host# set system services xnm-clear-text connection-limit limit
user@host# set system services xnm-clear-text rate-limit limit
```

For more information about the **xnm-clear-text** statement, see the *Junos OS System Basics Configuration Guide*.

4. Commit the configuration:

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

5. Repeat Step 2 through Step 4 on each device where the client application establishes Junos XML protocol sessions.

Prerequisites for SSH Connections

To enable a client application to use the SSH protocol to connect to the Junos XML protocol server, perform the following steps:

1. Enable the application to access the SSH software.

   If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSH software. For instructions, see "Downloading the Junos XML Protocol Perl Client and Prerequisites Package" on page 204.

   If the application does not use the Junos XML protocol Perl module, obtain the SSH software and install it on the computer where the application runs. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/ .

2. Satisfy the prerequisites discussed in "Prerequisites for All Access Protocols" on page 29.

3. (Optional) If you want to use key-based SSH authentication for the application, create a public/private key pair and associate it with the Junos OS login account you created in "Prerequisites for All Access Protocols" on page 29. Perform the following steps:

   a. Working on the computer where the client application runs, issue the **ssh-keygen** command in a standard command shell (not the Junos OS CLI). 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 man page provided by your SSH vendor for the **ssh-keygen** command. The Junos OS uses SSH version 2 by default but also supports version 1.

      ```
      % ssh-keygen options
      ```

b. Enable the application to access the public and private keys. One method is to run the **ssh-agent** program on the computer where the application runs.

c. On the device running Junos OS that needs to accept SSH connections from Junos XML protocol client applications, 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. First, move to that hierarchy level:

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

Issue the following command to copy 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*.

4. Configure the device running Junos OS to accept SSH connections by including the **ssh** statement at the **[edit system services]** hierarchy level. This statement enables SSH access for all users and applications, not just Junos XML protocol client applications.

```
[edit system login user account-name authentication]
user@host# top
[edit]
user@host# set system services ssh
```

5. Commit the configuration:

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

6. Repeat Step 1 on each computer where the application runs, and Step 2 through Step 5 on each device to which the application connects.

Prerequisites for Outbound SSH Connections

The outbound SSH feature allows the initiation of an SSH session between devices running Junos OS and Network and System Management servers where client-initiated TCP/IP connections are blocked (for example, when the device is behind a firewall). To configure outbound SSH, you add an **outbound-ssh** configuration statement to the device. Once configured and committed, the device running Junos OS will begin to initiate outbound SSH sessions with the configured management clients. Once the outbound SSH session is initialized and the connection is established, the management server

initiates the SSH sequence as the client and the device running Junos OS, acting as the server, authenticates the client.

Setting up outbound SSH involves:

- Configuring the device running Junos OS for outbound SSH
- Configuring the management server for outbound SSH.

To configure the device for outbound SSH:

1. Satisfy the prerequisites discussed in "Prerequisites for All Access Protocols" on page 29.
2. In the **[edit system services ssh]** hierarchy level, set the SSH protocol to v2:

   ```
   [edit system services ssh]
   set protocol-version v2
   ```

3. Generate/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*.
4. If the public key will be installed on the application management system manually, transfer the public key to the NSM server.
5. Add the following **outbound-ssh** statement at the **[edit system services]** hierarchy level:

   ```
   [edit system services]
   outbound-ssh client {
     application-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;
       }
     }
   }
   ```

   The attributes are as follows:

   - **application-id**—(Required) Identifies the **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**—(Required) Identifies the device to the client during the initiation sequence.

- **secret** ***secret***—(Optional) Public SSH host key of the device running the Junos OS. If this statement is added to the **outbound-ssh** configuration hierarchy, the device running Junos OS 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 router'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 establish 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 of the clients listed in the configuration management server list..

- **services**—(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.

6. Commit the configuration:

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

To set up the configuration management server:

1. Satisfy the prerequisites discussed in “Prerequisites for All Access Protocols” on page 29.
2. Enable the application to access the SSH software.
   - If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSH software. For instructions, see “Downloading the Junos XML Protocol Perl Client and Prerequisites Package” on page 204.
   - If the application does not use the Junos XML protocol Perl module, obtain the SSH software and install it on the computer where the application runs. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/ .
3. (Optional) Manually install the device's public key for use with the SSH connection.
4. Configure the client system to receive and process initialization broadcast requests. The intialization requests use the following syntax:
   - If the secret attribute is configured, the device running Junos OS will send its public SSH key along with the intialization sequence (recommended method). When the key has been received, the client needs to determine what to do with the device’s public key. We recommend that you replace any current public SSH key for the device with the new key. This ensures that the client always has the current key available for authentication.

     ```
     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
     ```

   - If the secret attribute is not configured, the device does not send its public SSH key along with the initialization sequence. You need to manually install the current public SSH key for the device.

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


Prerequisites for SSL Connections

To enable a client application to use the SSL protocol to connect to the Junos XML protocol server, perform the following steps:

1. Enable the application to access the SSL software.

   If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSL software. For instructions, see "Downloading the Junos XML Protocol Perl Client and Prerequisites Package" on page 204.

   If the application does not use the Junos XML protocol Perl module, obtain the SSL software and install it on the computer where the application runs. For information about obtaining and installing the SSL software, see http://www.openssl.org/ .

2. Satisfy the prerequisites discussed in "Prerequisites for All Access Protocols" on page 29.

3. Use one of the following two methods to obtain an authentication certificate in privacy-enhanced mail (PEM) format:

   - Request a certificate from a certificate authority; these agencies usually charge a fee.
   - Working on the computer where the client application runs, issue the following **openssl** command in a standard command shell (not the Junos OS CLI). The command generates a self-signed certificate and an unencrypted 1024-bit RSA private key, and writes them to the file called *certificate-file*.pem in the working directory. The command appears here on two lines only for legibility:

     **% openssl req -x509 -nodes -newkey rsa:1024 **
     -keyout *certificate-file*.pem -out *certificate-file*.pem

4. Import the certificate onto the device running Junos OS by including the **local** statement at the **[edit security certificates]** hierarchy level and the **load-key-file** statement at the **[edit security certificates local *certificate-name*]** hierarchy level.

   [edit]
   user@host# **edit security certificates local *certificate-name***

   [edit security certificates local *certificate-name*]
   user@host# **set load-key-file *URL-or-path***

   certificate-name is a name you choose to identify the certificate uniquely (for example, **junos-xml-protocol-ssl-client-*hostname***, where ***hostname*** is the computer where the client application runs).

   URL-or-path specifies the file that contains the paired certificate and private key (if you issued the **openssl** command in Step 3, the ***certificate-name*.pem** file). Specify either the URL to its location on the client computer or a pathname on the local disk (if you have already used another method to copy the certificate file to the device's local disk). For more information about specifying URLs and pathnames, see the *Junos OS CLI User Guide*.

NOTE: The CLI expects the private key in the *URL-or-path* file to be unencrypted. If the key is encrypted, the CLI prompts you for the passphrase associated with it, decrypts it, and stores the unencrypted version.

The set-load-key-file *URL-or-path* command copies the contents of the certificate file into the configuration. When you view the configuration, the CLI displays the string of characters that constitute the private key and certificate, marking them as SECRET-DATA. The load-key-file keyword is not recorded in the configuration.

5. Configure the device running Junos OS to accept SSL connections from Junos XML protocol client applications on port 3220 by including the **xnm-ssl** statement at the **[edit system services]** hierarchy level.

   ```
   [edit security certificates local certificate-name]
   user@host# top
   [edit]
   user@host# set system services xnm-ssl local-certificate certificate-name
   ```

   certificate-name is the unique name you assigned to the certificate in Step 4.

   By default, the Junos XML protocol server supports up to 75 simultaneous SSL sessions and 150 connection attempts per minute. Optionally, you can include either or both the **connection-limit** statement to limit the number of concurrent sessions and the **rate-limit** statement to limit connection attempts. Both statements accept a value from **1** through **250**.

   ```
   [edit]
   user@host# set system services xnm-ssl connection-limit limit
   user@host# set system services xnm-ssl rate-limit limit
   ```

   For more information about the **xnm-ssl** statement, see the *Junos OS System Basics Configuration Guide*.

6. Commit the configuration:

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

7. Repeat Step 1 on each computer where the client application runs, and Step 2 through Step 6 on each device to which the client application connects.

Prerequisites for Telnet Connections

To enable a client application to use the Telnet protocol to access the Junos XML protocol server, perform the steps described in this section.

Devices running the Junos-FIPS software do not accept Telnet connections. We recommend that you do not use the Telnet protocol in a Common Criteria environment. For more information, see the *Secure Configuration Guide for Common Criteria and Junos-FIPS*.

1. Verify that the application can access the Telnet software. On most operating systems, Telnet is accessible in the standard distribution.
2. Satisfy the prerequisites discussed in "Prerequisites for All Access Protocols" on page 29.
3. Configure the device running Junos OS to accept Telnet connections by including the **telnet** statement at the **[edit system services]** hierarchy level. This statement enables Telnet access for all users and applications, not just Junos XML protocol client applications.

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

4. Repeat Step 1 on each computer where the application runs, and Step 2 and Step 3 on each device to which the application connects.

Connecting to the Junos XML Protocol Server

Connecting to the Junos XML Protocol Server from the Client Application

For a client application to connect to the Junos XML protocol server and open a session, you must first satisfy the prerequisites described in "Prerequisites for Establishing a Connection" on page 29.

When the prerequisites are satisfied, an application written in Perl can most efficiently establish a connection and open a session by using the Junos XML protocol Perl module provided by Juniper Networks. For more information, see "Writing Junos XML Protocol Perl Client Applications" on page 203.

A client application that does not use the Junos XML protocol Perl module connects to the Junos XML protocol server by opening a socket or other communications channel to the Junos XML protocol server device, invoking one of the remote-connection routines appropriate for the programming language and access protocol that the application uses.

What the client application does next depends on which access protocol it is using:

- If using the clear-text or SSL protocol, the client application performs the following steps:

1. Emits the initialization PI and tag, as described in “Emitting the Initialization PI and Tag” on page 41.
2. Authenticates with the Junos XML protocol server, as described in “Authenticating with the Junos XML Protocol Server” on page 45.

- If using the SSH or Telnet protocol, the client application performs the following steps:
  1. Uses the protocol’s built-in authentication mechanism to authenticate.
  2. Issues the **junoscript** command to request that the Junos XML protocol server convert the connection into a Junos XML protocol session. For a C programming language example, see “Writing Junos XML Protocol C Client Applications” on page 231.
  3. Emits the initialization PI and tag, as described in “Emitting the Initialization PI and Tag” on page 41.

Connecting to the Junos XML Protocol Server from the CLI

The Junos XML management protocol and Junos XML API are primarily intended for use by client applications; however, for testing purposes you can establish an interactive Junos XML protocol session and type commands in a shell window. To connect to the Junos XML protocol server from the CLI operational mode, issue the **junoscript interactive** command (the **interactive** option causes the Junos XML protocol server to echo what you type):

```
user@host> junoscript interactive
```

To begin a Junos XML protocol session over the connection, emit the initialization PI and tag that are described in “Emitting the Initialization PI and Tag” on page 41. You can then type sequences of tag elements that represent operational and configuration operations, which are described in “Requesting Information” on page 65, “Changing Configuration Information” on page 111, and “Committing a Configuration” on page 141. To eliminate typing errors, save complete tag element sequences in a file and use a cut-and-paste utility to copy the sequences to the shell window.

NOTE: When you close the connection to the Junos XML protocol server (for example, by emitting the <request-end-session/> and </junoscript> tags), the routing platform completely closes your connection instead of returning you to the CLI operational mode prompt. For more information about ending a Junos XML protocol session, see “Ending a Junos XML Protocol Session and Closing the Connection” on page 58.

Starting the Junos XML Protocol Session

Each Junos XML protocol session begins with a handshake in which the Junos XML protocol server and the client application specify the version of XML and the version of the Junos XML management protocol they are using. Each party parses the version

information emitted by the other, using it to determine whether they can communicate successfully. The following sections describe how to start a Junos XML protocol session:

Emitting the Initialization PI and Tag

When the Junos XML protocol session begins, the client application emits an **<?xml?>** PI and an opening **<junoscript>** tag, as described in the following sections:

Emitting the <?xml?> PI

The client application begins by emitting an **<?xml?>** PI.

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.

```
<?xml version="version" encoding="encoding"?>
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see “Supported Software Versions” on page 45.

- **version**—The version of XML with which tag elements emitted by the client application comply
- **encoding**—The standardized character set that the client application uses and can understand

In the following example of a client application’s **<?xml?>** PI, the **version="1.0"** attribute indicates that the application is emitting tag elements that comply with the XML 1.0 specification. The **encoding="us-ascii"** attribute indicates that the client application is using the 7-bit ASCII character set standardized by the American National Standards Institute (ANSI). For more information about ANSI standards, see http://www.ansi.org/ .

```
<?xml version="1.0" encoding="us-ascii"?>
```

NOTE: If the application fails to emit the <?xml?> PI before emitting the opening <junoscript> tag, the Junos XML protocol server emits an error message and immediately closes the session and connection. For more information, see “Emitting the Opening <junoscript> Tag” on page 42.

Emitting the Opening <junoscript> Tag

The client application then emits its opening **<junoscript>** tag, which has the following syntax (and appears here on two lines only for legibility):

```
<junoscript version="version" hostname="hostname" junos:key="key"
  release="release-code">
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see "Supported Software Versions" on page 45.

- **version**—(Required) Specifies the version of the Junos XML management protocol that the client application is using.
- **hostname**—(Optional) Names the machine on which the client application is running. The information is used only when diagnosing problems. The Junos XML protocol does not include support for establishing trusted-host relationships or otherwise altering Junos XML protocol server behavior depending on the client hostname.
- **junos:key**—(Optional) Requests that the Junos XML protocol server indicate whether a child configuration element is an identifier for its parent element. The only acceptable value is **key**. For more information, see "Requesting an Indicator for Identifiers" on page 74.
- **release**—(Optional) Identifies the Junos OS Release (and by implication, the Junos XML API) for which the client application is designed. The value of this attribute indicates that the client application can interoperate successfully with a Junos XML protocol server that also supports that version of the Junos XML API. In other words, it indicates that the client application emits request tag elements from that API and knows how to parse response tag elements from it. If the application does not include this attribute, the Junos XML protocol server emits tag elements from the Junos XML API that it supports. For more information, see "Verifying Software Compatibility" on page 45.

  For the value of the **release** attribute, use the standard notation for Junos OS version numbers. For example, the value **10.4R1** represents the initial version of Junos OS Release 10.4.

In the following example of a client application's opening **<junoscript>** tag, the **version="1.0"** attribute indicates that it is using Junos XML protocol version 1.0. The **hostname="client1"** attribute indicates that the client application is running on the machine called **client1**. The **release="10.4R1"** attribute indicates that the switch, router, or security device is running the initial version of Junos OS Release 10.4.

```
<junoscript version="1.0" hostname="client1" release="10.4R1">
```


NOTE: If the application fails to emit the <?xml?> PI before emitting the opening <junoscript> tag, the Junos XML protocol server emits an error message similar to the following and immediately closes the session and connection:

```
<rpc-reply>
  <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <message>
      communication error while exchanging credentials
    </message>
  </xnm:error>
</rpc-reply>
<!-- session end at YYYY-MM-DD hh:mm:ss TZ -->
</junoscript>
```

For more information about the <xnm:error> tag, see "Handling an Error or Warning" on page 53.

Parsing the Initialization PI and Tag from the Junos XML Protocol Server

When the Junos XML protocol session begins, the Junos XML protocol server emits an **<?xml?>** PI and an opening **<junoscript>** tag, as described in the following sections:

Parsing the Junos XML Protocol Server's <?xml?> PI

The syntax for the **<?xml?>** PI is as follows:

```
<?xml version="version" encoding="encoding"?>
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see "Supported Software Versions" on page 45.

- **version**—The version of XML with which tag elements emitted by the Junos XML protocol server comply
- **encoding**—The standardized character set that the Junos XML protocol server uses and can understand

In the following example of a Junos XML protocol server's **<?xml?>** PI, the **version="1.0"** attribute indicates that the server is emitting tag elements that comply with the XML 1.0 specification. The **encoding="us-ascii"** attribute indicates that the server is using the 7-bit ASCII character set standardized by ANSI. For more information about ANSI standards, see http://www.ansi.org/ .

```
<?xml version="1.0" encoding="us-ascii"?>
```

Parsing the Junos XML Protocol Server's Opening <junoscript> Tag

After emitting the **<?xml?>**PI, the server then emits its opening **<junoscript>** tag, which has the following form (the tag appears on multiple lines only for legibility):

```
<junoscript xmlns="namespace-URL" xmlns:junos="namespace-URL" \
 schemaLocation="namespace-URL" os="JUNOS" \
 release="release-code" hostname="hostname" version="version">
```

The attributes are as follows:

- **hostname**—The name of the device on which the Junos XML protocol server is running.
- **os**—The operating system of the device on which the Junos XML protocol server is running. The value is always **JUNOS**.
- **release**—The identifier for the version of the Junos OS from which the Junos XML protocol server is derived and that it is designed to understand. It is presumably in use on the device where the Junos XML protocol server is running. The value of the **release** attribute uses the standard notation for Juniper Networks software version numbers. For example, the value **10.4R1** represents the initial version of Junos OS Release 10.4.
- **schemaLocation**—The XML namespace for the XML Schema-language representation of the Junos configuration hierarchy.
- **version**—The version of the Junos XML management protocol that the Junos XML protocol server is using.
- **xmlns**—The XML namespace for the tag elements enclosed by the **<junoscript>** tag element that do not have a prefix on their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.
- **xmlns:junos**—The XML namespace for the tag elements enclosed by the **<junoscript>** tag element that have the **junos:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/junos/*release-code*/junos**, where ***release-code*** is the standard string that represents a release of the Junos OS. For example, the value **10.4R1** represents the initial version of Junos OS Release 10.4.

In the following example of a Junos XML protocol server's opening **<junoscript>** tag, the **version** attribute indicates that the server is using Junos XML protocol version 1.0, and the **hostname** attribute indicates that the router's name is **big-device**. The **os** and **release** attributes indicate that the device is running the initial version of Junos OS Release 10.4. The **xmlns** attribute indicate that the default namespace for Junos XML tag elements is **http://xml.juniper.net/xnm/1.1/xnm** . The **xmlns:junos** attribute indicates that the namespace for tag elements that have the **junos:** prefix is **http://xml.juniper.net/junos/10.4R1/junos** . The tag appears on multiple lines only for legibility.

```
<junoscript xmlns="http://xml.juniper.net/xnm/1.1/xnm"\
   xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos" \
  schemaLocation="http://xml.juniper.net/junos/10.4R1/junos" os="JUNOS" \
  release="10.4R1.8" hostname="big-device" version="1.0">
```

Verifying Software Compatibility

Exchanging **<?xml?>** and **<junoscript>** tag elements enables a client application and the Junos XML protocol server to determine if they are running different versions of the software used during a Junos XML protocol session. Different versions are sometimes incompatible, and by Junos XML protocol convention the party running the later version of software determines how to handle any incompatibility. For fully automated performance, include code in the client application that determines if its version of software is later than that of the Junos XML protocol server. Decide which of the following options is appropriate when the application's version is more recent, and implement the corresponding response:

- Ignore differences in Junos version, and do not alter the client application's behavior to accommodate the Junos XML protocol server. A difference in Junos versions does not necessarily make the server and client incompatible, so this is often a valid approach.
- Alter standard behavior to be compatible with the Junos XML protocol server. If the client application is running a later version of the Junos OS, for example, it can choose to emit only tag elements that represent the software features available in the Junos XML protocol server's version of the Junos OS.
- End the Junos XML protocol session and terminate the connection. This is appropriate if you decide that it is not practical to accommodate the Junos XML protocol server's version of software. For instructions, see "Ending a Junos XML Protocol Session and Closing the Connection" on page 58.

Supported Software Versions

Table 6 on page 45 specifies the PI or opening tag and attribute used to convey version information during Junos XML protocol session initialization in version 1.0 of the Junos XML management protocol.

Table 6: Junos XML Protocol version 1.0 PI and Opening Tag

Software and Versions	PI or Tag	Attribute
XML 1.0	**<?xml?>**	**version="1.0"**
ANSI-standardized 7-bit ASCII character set	**<?xml?>**	**encoding="us-ascii"**
Junos XML protocol 1.0	**<junoscript>**	**version="1.0"**
Junos OS Release	**<junoscript>**	**release="m.nZb"** For example: **release="10.3R1"**

Authenticating with the Junos XML Protocol Server

A client application that uses the clear-text or SSL protocol must now authenticate with the Junos XML protocol server. (Applications that use the SSH or Telnet protocol use

the protocol's built-in authentication mechanism before emitting initialization tag elements, as described in "Connecting to the Junos XML Protocol Server" on page 39.)

See the following sections:

Submitting an Authentication Request

The client application begins the authentication process by emitting an **<rpc>** tag element enclosing the **<request-login>** tag element. In the **<request-login>** tag element, it encloses the **<username>** tag element to specify the Junos OS account (username) under which to establish the connection. The account must already be configured on the Junos XML protocol server device, as described in "Prerequisites for All Access Protocols" on page 29. You can choose whether or not the application provides the account password as part of the initial tag sequence.

Providing the Password with the Username

To provide the password along with the username, the application emits the following tag sequence:

```
<rpc>
  <request-login>
    <username>username</username>
    <challenge-response>password</challenge-response>
  </request-login>
</rpc>
```

This tag sequence is appropriate if the application automates access to routing, switching, or security platform information and does not interact with users, or obtains the password from a user before beginning the authentication process.

Providing Only the Username

To omit the password and specify only the username, the application emits the following tag sequence:

```
<rpc>
  <request-login>
    <username>username</username>
  </request-login>
</rpc>
```

This tag sequence is appropriate if the application does not obtain the password until the authentication process has already begun. In this case, the Junos XML protocol server returns the **<challenge>** tag element within an **<rpc-reply>** tag element to request the password associated with the username. The tag element encloses the **Password:** string, which the client application can forward to the screen as a prompt for a user. The **echo="no"** attribute in the opening **<challenge>** tag specifies that the password string typed by the user does not echo on the screen. The tag sequence is as follows:

```
<rpc-reply xmlns:junos="URL">
  <challenge echo="no">Password:</challenge>
</rpc-reply>
```

The client application obtains the password and emits the following tag sequence to forward it to the Junos XML protocol server:

```
<rpc>
  <request-login>
    <username>username</username>
    <challenge-response>password</challenge-response>
  </request-login>
</rpc>
```

Interpreting the Authentication Response

After it receives the username and password, the Junos XML protocol server emits the **<authentication-response>** tag element to indicate whether the authentication attempt is successful.

Server Response When Authentication Succeeds

If the password is correct, the authentication attempt succeeds and the Junos XML protocol server emits the following tag sequence:

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>success</status>
    <message>username</message>
    <login-name>remote-username</login-name>
  </authentication-response>
</rpc-reply>
```

The **<message>** tag element contains the Junos username under which the connection is established.

The **<login-name>** tag element contains the username that the client application provided to an authentication utility such as RADIUS or TACACS+. This tag element appears only if the username differs from the username contained in the **<message>** tag element.

The Junos XML protocol session begins, as described in “Starting the Junos XML Protocol Session” on page 40.

Server Response When Authentication Fails

If the password is not correct or the **<request-login>** tag element is otherwise malformed, the authentication attempt fails and the Junos XML protocol server emits the following tag sequence:

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>fail</status>
    <message>error-message</message>
  </authentication-response>
</rpc-reply>
```

The ***error-message*** string in the **<message>** tag element explains why the authentication attempt failed. The Junos XML protocol server emits the **<challenge>** tag element up to two more times before rejecting the authentication attempt and closing the connection.

Exchanging Information with the Junos XML Protocol Server

The session continues when the client application sends a request to the Junos XML protocol server. The Junos XML protocol 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 Junos XML Protocol Server

To initiate a request to the Junos XML protocol 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>
```

The application encloses each request in a separate pair of opening **<rpc>** and closing **</rpc>** tags. The **<rpc>** tag element can occur only within the **<junoscript>** tag element. For an example of emitting an **<rpc>** tag element in the context of a complete Junos XML protocol session, see "Example of a Junos XML Protocol Session" on page 60.

The Junos XML protocol server ignores any newline characters, spaces, or other white space characters that occur between tag elements in the tag stream, but does preserve 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:

NOTE: Although operational and configuration requests conceptually belong to separate classes, a Junos XML protocol 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 66. 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 Junos XML protocol defines the **<get-configuration>** 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 about the **[edit system login]** hierarchy level in the candidate configuration:

```
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

For more information, see “Requesting Configuration Information” on page 68. For a summary of Junos XML 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 Junos XML protocol defines the **<load-configuration>** operation 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>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <user>
            <name>admin</name>
            <full-name>Administrator</full-name>
            <class>superuser</class>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

For more information, see “Changing Configuration Information” on page 111 and “Committing a Configuration” on page 141. 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 Junos XML protocol 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 Junos XML protocol server echoes the attribute in its opening **<rpc-reply>** tag, making it easy to map the response to the initiating request. The client application can freely define attribute names, except as described in the following note.

NOTE: The xmlns:junos attribute name is reserved. The Junos XML protocol server sets the attribute to an appropriate value on the opening <rpc-reply> tag, so client applications must not emit it on the opening <rpc> tag. For more information, see “xmlns:junos Attribute” on page 51.

Parsing the Junos XML Protocol Server Response

The Junos XML protocol server encloses its response to each client request in a separate pair of opening **<rpc-reply>** and closing **</rpc-reply>** tags. Each response constitutes a well-formed XML document.

```
<rpc-reply xmlns:junos="URL">
  <!-- tag elements representing a response -->
</rpc-reply>
```

The **xmlns:junos** attribute in the opening **<rpc-reply>** tag defines the default namespace for the enclosed tag elements that have the **junos:** prefix in their names, as discussed further in "xmlns:junos Attribute" on page 51. The **<rpc-reply>** tag element occurs only within the **<junoscript>** tag element. Client applications must include code for parsing the stream of response tag elements coming from the Junos XML protocol server, either processing them as they arrive or storing them until the response is complete. See the following sections for further information:

For an example of parsing the **<rpc-reply>** tag element in the context of a complete Junos XML protocol session, see "Example of a Junos XML Protocol Session" on page 60.

xmlns:junos Attribute

The Junos XML protocol server includes the **xmlns:junos** attribute in the opening **<rpc-reply>** tag to define the XML namespace for the enclosed Junos XML tag elements that have the **junos:** prefix on their names. The namespace is a URL of the form **http://xml.juniper.net/junos/*release-code*/junos**, where ***release-code*** is the standard string that represents the release of the Junos OS running on the Junos XML protocol server machine.

In the following example, the namespace corresponds to the initial version of Junos OS Release 10.4:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
```

Junos XML Protocol Server Response Classes

The Junos XML protocol 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 Junos XML protocol 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 contains interface information for the initial version of Junos OS Release 10.4.

```
<rpc-reply 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 contents of operational response tag elements, see “Requesting Operational Information” on page 66. 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.

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <system>
      <login>
        <user>
          <name>admin</name>
          <full-name>Administrator</full-name>
          <!-- other data tag elements for the admin user -->
        </user>
      </login>
    </system>
  </configuration>
</rpc-reply>
```

Configuration Change Responses

Configuration change responses are responses to requests that change the state or contents of the device configuration. For commit operations, the Junos XML protocol server encloses an explicit indicator of success or failure within the **<commit-results>** tag element:

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <!-- tag elements for information about the commit -->
```



```
    </commit-results>
</rpc-reply>
```

For other operations, the Junos XML protocol server indicates success by returning an opening **<rpc-reply>** and closing **</rpc-reply>** tag with nothing between them, instead of emitting an explicit success indicator:

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

For more information, see "Changing Configuration Information" on page 111 and "Committing a Configuration" on page 141. 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 implements 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 Junos XML protocol server encounters an error condition, it emits an **<xnm:error>** tag element, which encloses child tag elements that describe the nature of the error. Client applications must be prepared to receive and handle an **<xnm: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.

The syntax of the **<xnm:error>** tag element is as follows. The opening tag appears on multiple lines only for legibility:

```
<xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <!-- tag elements describing the error -->
</xnm:error>
```

The attributes are as follows:

- **xmlns**—The XML namespace for the tag elements enclosed by the **<xnm:error>** tag element that do not have a prefix in their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.
- **xmlns:xnm**—The XML namespace for the **<xnm:error>** tag element and for the enclosed tag elements that have the **xnm:** prefix in their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

The set of child tag elements enclosed in the **<xnm:error>** tag element depends on the operation that server was performing when the error occurred. 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 (discussed in "Requesting Information" on page 65)
- Opening, locking, committing, or closing a configuration as requested by a client application (discussed in "Exchanging Information with the Junos XML Protocol Server" on page 48, "Committing a Configuration" on page 141, and "Ending a Junos XML Protocol Session and Closing the Connection" on page 58)
- Parsing configuration data submitted by a client application in a **<load-configuration>** tag element (discussed in "Changing Configuration Information" on page 111)

If the Junos XML protocol server encounters a less serious problem, it can emit an **<xnm:warning>** tag element instead. The usual response for the client application in this case is to log the warning or pass it to the user, but to continue parsing the server's response.

For a description of the child tag elements that can appear within an **<xnm:error>** or **<xnm:warning>** tag element to specify the nature of the problem, see "<xnm:error>" on page 181 and "<xnm:warning>" on page 183.

Halting a Request

To request that the Junos XML protocol server stop processing the current request, a client application emits the **<abort/>** tag directly after the closing **</rpc>** tag for the operation to be halted:

```
<rpc>
  <!-- tag elements for the request -->
</rpc>
<abort/>
```

The Junos XML protocol server responds with the **<abort-acknowledgement/>** tag:

```
<rpc-reply xmlns:junos="URL">
  <abort-acknowledgement/>
</rpc-reply>
```


Depending on the operation being performed, response tag elements already sent by the Junos XML protocol server for the halted request are possibly invalid. The application can include logic for deciding whether to discard or retain them as appropriate.

For more information, see "<abort/>" on page 157 and "<abort-acknowledgement/>" on page 157.

Locking and Unlocking the Candidate Configuration or Creating a Private Copy

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

- Lock the candidate configuration, which prevents other users or applications from changing the configuration until the application releases the lock (equivalent to the CLI **configure exclusive** command).
- Create a private copy of the candidate configuration, which enables the application to view or change configuration data without affecting the candidate or active configuration until the private copy is committed (equivalent to the CLI **configure private** 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 or creating a private copy is not required. However, it is appropriate to lock the configuration if it is important that the information being returned not change during the session. The information from a private copy is guaranteed not to change, but can diverge from the candidate configuration if other users or applications are changing the candidate.

The restrictions on, and interactions between, operations on the locked regular candidate configuration and a private copy are the same as for the CLI **configure exclusive** and **configure private** commands. For more information, see "Committing a Private Copy of the Configuration" on page 143 and the *Junos OS CLI User Guide*.

For more information about locking and unlocking the candidate configuration or creating a private copy, see the following sections:

Locking the Candidate Configuration

To lock the candidate configuration, a client application emits the **<lock-configuration/>** tag within an **<rpc>** tag element:

```
<rpc>
  <lock-configuration/>
```



```
</rpc>
```

Emitting this tag 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 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.

When the Junos XML protocol server locks the configuration, it returns an opening **<rpc-reply>** and closing **</rpc-reply>** tag with nothing between them:

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

If the Junos XML protocol server cannot lock the configuration, the **<rpc-reply>** tag element instead encloses an **<xnm: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 login identity of the user or application.
- The candidate configuration already includes changes that have not yet been committed. To commit the changes, see “Committing a Configuration” on page 141. To roll back to a previous version of the configuration (and lose the uncommitted changes), see “Rolling Back to a Previous or Rescue Configuration” on page 117.

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, or can change their private copies. The lock persists until either the Junos XML protocol session ends or the client application unlocks the configuration by emitting the **<unlock-configuration/>** tag, as described in “Unlocking the Candidate Configuration” on page 56.

If the candidate configuration is not committed before the client application unlocks it, or if the Junos XML protocol 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-configuration/>** tag in an **<rpc>** tag element:

```
<rpc>
  <unlock-configuration/>
</rpc>
```

To confirm that it has successfully unlocked the configuration, the Junos XML protocol server returns an opening **<rpc-reply>** and closing **</rpc-reply>** tag with nothing between them:

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

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

Terminating Another Junos XML Protocol 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 55. In this case, the Junos XML protocol 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">
  <xnm:error>
    <message>
      configuration database locked by:
user terminal (pid PID) on since YYYY-MM-DD hh:mm:ss TZ, idle hh:mm:ss
      exclusive [edit]
    </message>
  </xnm: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 Junos XML protocol 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 Junos XML protocol 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 Junos XML protocol 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 145.)

Creating a Private Copy of the Configuration

To create a private copy of the candidate configuration, a client application emits the **<private/>** tag enclosed in **<rpc>** and **<open-configuration>** tag elements:

```
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>
</rpc>
```

The client application can then perform the same operations on the private copy as on the regular candidate configuration, as described in "Changing Configuration Information" on page 111.

After making changes to the private copy, the client application can commit the changes to the active configuration on the device running Junos OS by emitting the **<commit-configuration>** tag element, as for the regular candidate configuration. However, there are some restrictions on the commit operation for a private copy. For more information, see "Committing a Private Copy of the Configuration" on page 143.

To discard the private copy without committing it, a client application emits the **<close-configuration/>** tag enclosed in an **<rpc>** tag element:

```
<rpc>
  <close-configuration/>
</rpc>
```

Any changes to the private copy are lost. Changes to the private copy are also lost if the Junos XML protocol session ends for any reason before the changes are committed. It is not possible to save changes to a private copy other than by emitting the **<commit-configuration>** tag element.

The following example shows how to create a private copy of the configuration. The Junos XML protocol server includes a reminder in its confirmation response that changes are discarded from a private copy if they are not committed before the session ends.

Client Application

```
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <xnm:warning xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
               xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <message>uncommitted changes will be discarded on exit</message>
  </xnm:warning>
</rpc-reply>
```

T1172

Ending a Junos XML Protocol Session and Closing the Connection

When a client application is finished making requests, it ends the Junos XML protocol session by emitting the **<request-end-session/>** tag within an **<rpc>** tag element:

```
<rpc>
```

```
    <request-end-session/>
  </rpc>
```

In response, the Junos XML protocol server emits the **<end-session/>** tag enclosed in an **<rpc-reply>** tag element and a closing **</junoscript>** tag:

```
  <rpc-reply xmlns:junos="URL">
    <end-session/>
  </rpc-reply>
  </junoscript>
```

The client application waits to receive this reply before emitting its closing **</junoscript>** tag:

```
  </junoscript>
```

For an example of the exchange of closing tags, see "Closing the Junos XML Protocol Session" on page 63.

The client application can then close the SSH, SSL, or other connection to the Junos XML protocol server machine. Client applications written in Perl can close the Junos XML protocol session and connection by using the Junos XML protocol Perl module described in "Writing Junos XML Protocol Perl Client Applications" on page 203. For more information, see that chapter.

Client applications that do not use the Junos XML protocol Perl module use the routine provided for closing a connection in the standard library for their programming language.

Displaying CLI Output as XML Tag Elements

To display the output from a CLI command as Junos XML protocol tag elements and Junos XML tag elements instead of as the default formatted ASCII text, pipe the output from the command to the **display xml** command. 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 the Junos 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 only for legibility):

```
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 Junos XML Protocol Session

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

Exchanging Initialization PIs and Tag Elements

After the client application establishes a connection to a Junos XML protocol server, the two exchange initialization PIs and tag elements, as shown in the following example. Note that the Junos XML protocol server's opening **<junoscript>** tag appears on multiple lines for legibility only. Neither the Junos XML protocol server nor client applications insert a newline character into the list of attributes. Also, in an actual exchange, the ***JUNOS-release*** variable is replaced by a value such as **10.4R1** for the initial version of Junos OS Release 10.4. For a detailed discussion of the **<?xml?>** PI and opening **<junoscript>** tag, see "Starting the Junos XML Protocol Session" on page 40.

Client Application

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" release=" JUNOS-release">
```

Junos XML Protocol Server

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" hostname="router1" \
   os="JUNOS" release="JUNOS-release" \
   xmlns="URL" xmlns:junos="URL" \
   xmlns:xnm="URL">
```

T1173

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 Junos XML protocol server returns the requested information in the **<chassis-inventory>** tag element.

Client Application

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

Junos XML Protocol Server

```
<rpc-reply 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>
```

T1102

Locking the Configuration

The client application then prepares to create a new privilege class called **network-mgmt** at the **[edit system login class]** hierarchy level. It begins by using the **<lock-configuration/>** 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 Junos XML protocol server returns an **<rpc-reply>** and an **</rpc-reply>** tag with nothing between them.

Client Application

```
<rpc>
   <lock-configuration/>
</rpc>
```

Junos XML Protocol Server

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

T1103

Changing the Configuration

The client application emits the tag elements that add the new **network-mgmt** privilege class to the candidate configuration. The Junos XML protocol server returns the **<load-configuration-results>** tag element to enclose a tag element that reports the outcome of the load operation. (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 111.)

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>network-mgmt</name>
            <permissions>configure</permissions>
            <permissions>snmp</permissions>
            <permissions>system</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1104

Committing the Configuration

The client application commits the candidate configuration. The Junos XML protocol server returns the **<commit-results>** tag element to enclose tag elements that report the outcome of the commit operation (for information about these tag elements, see “Committing a Configuration” on page 141).

Client Application

```
<rpc>
  <commit-configuration/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1105

Unlocking the Configuration

The client application unlocks (and by implication closes) the candidate configuration. As when it opens the configuration, the Junos XML protocol server confirms successful closure of the configuration only by returning an opening **<rpc-reply>** and closing **</rpc-reply>** tag with nothing between them.

Client Application	Junos XML Protocol Server
<rpc> <unlock-configuration/> </rpc>	
	<rpc-reply xmlns:junos="URL"> </rpc-reply>

T1106

Closing the Junos XML Protocol Session

The client application closes the Junos XML protocol session.

Client Application	Junos XML Protocol Server
<rpc> <request-end-session/> </rpc>	
	<rpc-reply xmlns:junos="URL"> <end-session/> </rpc-reply> </junoscript>
</junoscript>	

T1165

CHAPTER 4

Requesting Information

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

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 Junos XML protocol **<get-configuration>** 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 Junos XML protocol 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 Junos XML protocol server, a client application performs the procedures described in the indicated sections:

1. Establishes a connection to the Junos XML protocol server on the routing, switching, or security platform, as described in "Connecting to the Junos XML Protocol Server" on page 39.
2. Opens a Junos XML protocol session, as described in "Starting the Junos XML Protocol Session" on page 40.
3. If making configuration requests, optionally locks the candidate configuration or creates a private copy, as described in "Locking the Candidate Configuration" on page 55 and "Creating a Private Copy of the Configuration" on page 58.
4. Makes any number of requests one at a time, freely intermingling operational and configuration requests. See "Requesting Operational Information" on page 66 and "Requesting Configuration Information" on page 68.

   The application can also intermix requests with configuration changes, which are described in "Changing Configuration Information" on page 111.
5. Accepts the tag stream emitted by the Junos XML protocol server in response to each request and extracts its content, as described in "Parsing the Junos XML Protocol Server Response" on page 51.
6. Unlocks the candidate configuration if it is locked, as described in "Unlocking the Candidate Configuration" on page 56. Other users and applications cannot change the configuration while it remains locked.
7. Ends the Junos XML protocol session and closes the connection to the device, as described in "Ending a Junos XML Protocol Session and Closing the Connection" on page 58.

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 Junos XML protocol 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:junos="URL">
  <operational-response xmlns="URL-for-DTD">
    <!-- Junos XML 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 Junos XML protocol 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 Junos XML protocol 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

For a reference page for the **<output>** tag element, see the *Junos XML API Operational Reference*.

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-configuration>** tag element in an **<rpc>** tag element. By setting optional attributes, the client application can specify the source and formatting of the configuration information returned by the Junos XML protocol server. By including the appropriate optional child tag elements, the application can request the entire configuration or specific portions of the configuration. The basic syntax is as follows:

```
<rpc>
  <!-- If requesting the complete configuration -->
    <get-configuration [optional attributes]/>

  <!-- If requesting part of the configuration -->
    <get-configuration [optional attributes]>
      <configuration>
        <!-- tag elements representing the data to return -->
      </configuration>
    </get-configuration>
</rpc>
```

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 “Exchanging Information with the Junos XML Protocol Server” on page 48.

The Junos XML protocol server encloses its reply in an **<rpc-reply>** tag element. It includes attributes with the **junos:** prefix in the opening **<configuration>** tag to indicate when the configuration was last changed or committed and who committed it (the attributes appear on multiple lines in the syntax statement only for legibility). For more information about them, see “Requesting Information from the Committed or Candidate Configuration” on page 70:

```
<rpc-reply xmlns:junos="URL">
  <!-- If the application requests Junos XML tag elements -->
  <configuration junos:(changed | commit)-seconds="seconds" \
      junos:(changed | commit)-localtime="YYYY-MM-DD hh:mm:ss TZ" \
      [junos:commit-user="username"]>
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>

  <!-- If the application requests formatted ASCII text -->
```

```
    <configuration-text>
      <!-- formatted ASCII configuration statements -->
    </configuration-text>
  </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.

For reference pages for the **<configuration>**, **<configuration-text>**, and **<undocumented>** tag elements, see the *Junos XML API Operational Reference*.

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 103
- Requesting a Previous (Rollback) Configuration on page 106
- Comparing Two Previous (Rollback) Configurations on page 108
- Requesting the Rescue Configuration on page 109

The following sections describe how a client application specifies the source, format, and amount of information returned by the Junos XML protocol server:

- Specifying the Source and Output Format of Configuration Information on page 69
- Specifying the Scope of Configuration Information to Return on page 89

Specifying the Source and Output Format of Configuration Information

By including optional attributes when requesting configuration information, a client application can specify the source and formatting of the output returned by the Junos XML protocol server, as described in the following sections:

- Requesting Information from the Committed or Candidate Configuration on page 70
- Requesting Output as Formatted ASCII Text or Junos XML Tag Elements on page 72
- Requesting an Indicator for Identifiers on page 74
- Requesting a Change Indicator for Configuration Elements on page 76
- Displaying Commit-Script-Style XML Data on page 79
- Specifying the Output Format for Configuration Groups and Interface Ranges on page 80
- Comparing Configuration Changes with a Prior Version on page 88

Requesting Information from the Committed or Candidate Configuration

To request information from the candidate configuration, the application either includes the **database="candidate"** attribute or omits the attribute completely (information from the candidate configuration is the default):

```
<rpc>
  <get-configuration/>

<!-- OR -->

  <get-configuration>
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To request information from the active configuration—the one most recently committed on the device—a client application includes the **database="committed"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

```
<rpc>
  <get-configuration database="committed"/>

<!-- OR -->

  <get-configuration database="committed">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Information to Return” on page 89.

The Junos XML protocol server encloses its response in the **<rpc-reply>** tag element and either the **<configuration>** tag element (for Junos XML-tagged output) or **<configuration-text>** tag element (for formatted ASCII output).

When returning information from the candidate configuration as Junos XML tag elements, the Junos XML protocol server includes attributes in the opening **<configuration>** tag that indicate when the configuration last changed (they appear on multiple lines here only for legibility):

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ">
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>
</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 as Junos XML tag elements, the Junos XML protocol server includes attributes in the opening **<configuration>** tag that indicate when the configuration was committed (they appear on multiple lines here only for legibility):

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:commit-seconds="seconds" \
    junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
    junos:commit-user="username">
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>
</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.

The **database** attribute in the application's request can be combined with one or more of the following attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in "Requesting a Change Indicator for Configuration Elements" on page 76
- **commit-scripts**, which is described in "Displaying Commit-Script-Style XML Data" on page 79
- **compare**, which is described in "Comparing Configuration Changes with a Prior Version" on page 88
- **format**, which is described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72
- **inherit** and optionally **groups** and **interface-ranges**, which are described in "Specifying the Output Format for Configuration Groups and Interface Ranges" on page 80

The application can also include the **database** attribute after requesting an indicator for identifiers (as described in "Requesting an Indicator for Identifiers" on page 74).

The following example shows how to request the entire committed configuration. 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 **Junos XML Protocol Server**

```
<rpc>
   <get-configuration database="committed"/>
</rpc>
```

```
<rpc-reply xmlns:junos='URL">
   <configuration \
         junos:commit-seconds="seconds" \
         junos:commit-localtime="timestamp" \
         junos:commit-user="username">
      <version>Junos-version</version>
      <system>
         <host-name>big-router</host-name>
         <!- - other children of <system>- ->
      </system>
      <!- - other children of <configuration>- ->
   </configuration>
</rpc-reply>
```

T1185

Requesting Output as Formatted ASCII Text or Junos XML Tag Elements

To request that the Junos XML protocol server return configuration information in Junos XML-tagged output, the client application either includes the **format="xml"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag or omits the attribute completely. The Junos XML protocol server returns Junos XML-tagged output by default, except when the **compare** attribute is included.

```
<rpc>
    <get-configuration/>

  <!-- OR -->

    <get-configuration>
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

To request that the Junos XML protocol server return configuration information as formatted ASCII text instead of tagging it with Junos XML tag elements, the client application includes the **format="text"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
    <get-configuration format="text"/>

  <!-- OR -->

    <get-configuration format="text">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Information to Return” on page 89.

NOTE: Regardless of which output format they request, client applications use Junos XML tag elements to represent the configuration element to display. The **format** attribute controls the format of the Junos XML protocol server's output only.

When the application requests Junos XML tag elements, the Junos XML protocol server encloses its output in **<rpc-reply>** and **<configuration>** tag elements. For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 70.

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

When the application requests formatted ASCII output, the Junos XML protocol server formats its response in the same way that the CLI **show configuration** command displays configuration data—it uses the newline character, tabs, braces, and square brackets to indicate the hierarchical relationships between configuration statements. The server encloses formatted ASCII configuration statements in **<rpc-reply>** and **<configuration-text>** tag elements:

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
    <!-- formatted ASCII configuration statements -->
  </configuration-text>
</rpc-reply>
```

The **format** attribute can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70
- **inherit** and optionally **groups** and **interface-ranges**, which are described in "Specifying the Output Format for Configuration Groups and Interface Ranges" on page 80

It does not make sense to combine the **format="text"** attribute with the **changed** attribute (described in "Requesting a Change Indicator for Configuration Elements" on page 76) or to include it after requesting an indicator for identifiers (described in "Requesting an Indicator for Identifiers" on page 74). The change and identifier indicators appear only in Junos XML-tagged output, which is the default output format. The **commit scripts** attribute returns Junos XML-tagged output by default, even if the **format="text"** attribute is included, since this is the format that is input to commit script. The **format="xml"** attribute cannot be used with the **compare** attribute, which produces only formatted ASCII output.

An application can request Junos-XML tagged output or formatted ASCII text for the entire configuration or any portion of it. For instructions on specifying the amount of data to return, see "Specifying the Scope of Configuration Information to Return" on page 89.

The following example shows how to request formatted ASCII output from the **[edit policy-options]** hierarchy level in the candidate configuration.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration format="text">
    <configuration>
      <policy-options/>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <configuration-text>
    policy-options {
      policy-statement load-balancing-policy {
        from {
          route-filter 192.168.10/24 orlonger;
          route-filter 10.114/16 orlonger;
        }
        then {
          load-balance per-packet;
        }
      }
    }
  </configuration-text>
</rpc-reply>
```

T1121

Requesting an Indicator for Identifiers

To request that the Junos XML protocol server indicate whether a child configuration element is an identifier for its parent element, a client application includes the **junos:key="key"** attribute in the opening **<junoscript>** tag for the Junos XML protocol session, which appears here on two lines for legibility only:

```
<junoscript version="version" hostname="hostname" junos:key="key"
            release="release-code">
```

For more information about the **<junoscript>** tag, see “Emitting the Opening <junoscript> Tag” on page 42.

When the identifier indicator is requested, the Junos XML protocol server includes the **junos:key="key"** attribute in the opening tag for each identifier. As always, the Junos XML protocol server encloses its response in **<rpc-reply>** and **<configuration>** tag elements. For information about the attributes in the opening **<configuration>** tag, see “Requesting Information from the Committed or Candidate Configuration” on page 70. In the following, the identifier tag element is called **<name>**:

```
<rpc-reply xmlns:junos="URL">
    <configuration attributes>
      <!-- opening tag for each parent of the object -->

  <!-- For each configuration object with an identifier -->
      <object>
        <name junos:key="key">identifier</name>
        <!-- additional children of object -->
        </object>
      <!-- closing tag for each parent of the object -->

  </configuration>
</rpc-reply>
```

The client application can include one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag when the **junos:key** attribute is included in the opening **<junoscript>** tag:

- **changed**, which is described in "Requesting a Change Indicator for Configuration Elements" on page 76
- **commit-scripts**, which is described in "Displaying Commit-Script-Style XML Data" on page 79
- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70
- **inherit** and optionally **groups** and **interface-ranges**, which are described in "Specifying the Output Format for Configuration Groups and Interface Ranges" on page 80

When requesting an indicator for identifiers, it does not make sense to include the **format="text"** attribute in the **<get-configuration>** tag element (as described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72). The **junos:key="key"** attribute appears only in Junos XML-tagged output, which is the default output format. The **compare** attribute produces only text output, so when this attribute is included in the **<get-configuration>** tag, the **junos:key="key"** attribute does not appear in the output.

The following example shows how indicators for identifiers appear on configuration elements at the **[edit interfaces]** hierarchy level in the candidate configuration when the **junos:key="key"** attribute is included in the opening **<junoscript>** tag emitted by the client application for the session. The two opening **<junoscript>** tags appear on multiple lines for legibility only. Neither client applications nor the Junos XML protocol server insert newline characters within tags. Also, for brevity the output includes just one interface, the loopback interface **lo0**.

Client Application

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" \
                junos:key="key" \
                release="JUNOS-release">

<rpc>
  <get-configuration>
    <configuration>
      <interfaces/>
    </configuration>
  </get-configuration>
</rpc>
```

Junos XML Protocol Server

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" hostname="router1" \
                os="JUNOS" release="JUNOS-release" \
                xmlns="URL" xmlns:junos="URL" \
                xmlns:xnm="URL">

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
       junos:changed-localtime="timestamp">
    <interfaces>
      <!- -tag elements for other interfaces- ->
      <interface>
        <name junos:key="key">lo0</name>
        <unit>
          <name junos:key="key">0</name>
          <family>
            <inet>
              <address>
                <name junos:key="key">127.0.0.1/32</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
      <!- -tag elements for other interfaces- ->
    </interfaces>
  </configuration>
</rpc-reply>
```

T1187

Requesting a Change Indicator for Configuration Elements

To request that the Junos XML protocol server indicate which configuration elements have changed since the last commit, a client application includes the **changed="changed"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
    <get-configuration changed="changed"/>

  <!-- OR -->

    <get-configuration changed="changed">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Information to Return” on page 89.

The Junos XML protocol server indicates which elements have changed by including the **junos:changed="changed"** attribute in the opening tag of every parent tag element in the path to the changed configuration element. If the changed configuration element is represented by a single (empty) tag, the **junos:changed="changed"** attribute appears in the tag. If the changed element is represented by a container tag element, the

junos:changed="changed" attribute appears in the opening container tag and also in the opening tag for each child tag element enclosed in the container tag element.

The Junos XML protocol server encloses its response in **<rpc-reply>** and **<configuration>** tag elements. For information about the standard attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 70.

```
<rpc-reply xmlns:junos="URL">
    <configuration standard-attributes junos:changed="changed">
      <!-- opening-tag-for-each-parent-level junos:changed="changed" -->

  <!-- For each changed element, EITHER -->
    <element junos:changed="changed"/>

  <!-- OR -->

    <element junos:changed="changed">
      <first-child-of-element junos:changed="changed">
      <second-child-of-element junos:changed="changed">
      <!-- additional children of element -->
    </element>

    <!-- closing-tag-for-each-parent-level -->
  </configuration>
</rpc-reply>
```

NOTE: When a commit operation succeeds, the Junos XML protocol server removes the junos:changed="changed" attribute from all tag elements. However, if warnings are generated during the commit, the attribute is not removed. In this case, the junos:changed="changed" attribute appears on tag elements that changed before the commit as well as those that changed after the commit.

An example of a commit-time warning is the message explaining that a configuration element will not actually apply until the device is rebooted. The warning appears in the tag string that the Junos XML protocol server returns to confirm the success of the commit, enclosed in an <xnm:warning> tag element.

To remove the junos:changed="changed" attribute from elements that changed before the commit, the client application must take any action necessary to eliminate the cause of the warning, and commit the configuration again.

The **changed** attribute can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70. Request change indicators in either the candidate or active configuration:

  - When the **database="candidate"** attribute is included or the **database** attribute is omitted, the output is from the candidate configuration. Elements added to the candidate configuration after the last commit operation are marked with the **junos:changed="changed"** attribute.
  - When the **database="committed"** attribute is included, the output is from the active configuration. Elements added to the active configuration by the most recent commit are marked with the **junos:changed="changed"** attribute.

- **inherit** and optionally **groups** and **interface-ranges**, which are described in "Specifying the Output Format for Configuration Groups and Interface Ranges" on page 80.

It does not make sense to combine the **changed** attribute with the **format="text"** attribute or with the **compare** attribute, which produces only text output. The **junos:changed="changed"** attribute appears only in Junos XML-tagged output, which is the default output format. When the **commit-scripts="view"** attribute is included in the **<get-configuration>** tag, the **junos:changed="changed"** attribute is automatically included in the output, and you do not need to explicitly include this attribute in the **<get-configuration>** request.

The application can also include the **changed** attribute after requesting an indicator for identifiers (as described in "Requesting an Indicator for Identifiers" on page 74).

The following example shows how to request change indicators for configuration elements at the **[edit system syslog]** hierarchy level in the candidate configuration. The output indicates that a log file called **interactive-commands** has been configured since the last commit.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration changed="changed">
    <configuration>
      <system>
        <syslog/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <configuration junos:changed-seconds='seconds' \
      junos:changed-localtime="timestamp" junos:changed="changed">
    <system junos:changed="changed">
      <syslog junos:changed="changed">
        <file>
          <name>messages</name>
          <contents>
            <name>any</name>
            <info/>
          </contents>
        </file>
        <file junos:changed="changed">
          <name junos:changed="changed">interactive-commands</name>
          <contents>
            <name junos:changed="changed">interactive-commands</name>
            <notice junos:changed="changed"/>
          </contents>
        </file>
      </syslog>
    </system>
  </configuration>
</rpc-reply>
```

T1186

Displaying Commit-Script-Style XML Data

To view the device's current configuration in Extensible Markup Language (XML) using the command-line interface's (CLI) operational mode, you issue the **show configuration | display xml** command. To view your configuration in commit-script-style XML, you use the **show configuration | display commit-scripts view** command. This command displays the configuration in the format that would be input to a commit script.

To request that the Junos XML protocol server display the configuration as commit-script-style XML data, a client application includes the **commit-scripts="view"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
    <get-configuration commit-scripts="view"/>

  <!-- OR -->

    <get-configuration commit-scripts="view">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

The **commit-scripts**, attribute can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70.
- **interface-ranges**, which is described in "Specifying the Output Format for Configuration Groups and Interface Ranges" on page 80.

You do not need to include the **changed** or the **inherit** attributes with the **commit-scripts="view"** attribute. The commit-scripts-style XML view includes the **junos:changed="changed"** attribute in the XML tags, and it displays the output with inheritance. That is, the output displays tag elements inherited from user-defined groups or interface ranges within the inheriting tag elements, and the XML tags already include the **junos:group** attribute. To explicitly display the **junos:interface-range** attribute in the commit-scripts-style view, you must include the **interface-ranges="interface-ranges"** attribute in the **<get-configuration>** tag.

Specifying the Output Format for Configuration Groups and Interface Ranges

The **<groups>** tag element corresponds to the **[edit groups]** configuration hierarchy. It encloses tag elements representing *configuration groups*, each of which contains a set of configuration statements that are appropriate at multiple locations in the hierarchy. Use the **apply-groups** configuration statement or **<apply-groups>** tag element to insert a configuration group at the appropriate location, achieving the same effect as directly inserting the statements defined in the group. The section of configuration hierarchy to which a configuration group is applied is said to *inherit* the group's statements.

In addition to the groups defined at the **[edit groups]** hierarchy level, the Junos OS predefines a group called **junos-defaults**. This group includes configuration statements judged appropriate for basic operations on any routing, switching, or security platform. By default, the statements in this group do not appear in the output of CLI commands that display the configuration, nor in the output returned by the Junos XML protocol server for the **<get-configuration>** tag element. For more information about user-defined configuration groups and the **junos-defaults** group, see the *Junos OS CLI User Guide*.

The **<interface-range>** tag element corresponds to the **[edit interfaces interface-range]** configuration hierarchy. When you configure an interface range, you specify a set of identical interfaces as an interface group, to which you can apply a common configuration to the entire set of interfaces. If an interface is a member of an interface range, it inherits the configuration statements set for that range.

The following sections explain how to specify the output format for configuration elements that are defined in configuration groups or interface ranges:

Specifying Whether Configuration Groups and Interface Ranges Are Displayed Separately

By default, the Junos XML protocol server displays the tag element for each user-defined configuration group as a child of the **<groups>** tag element, instead of displaying them as children of the elements to which they are applied. Similarly, the server displays the tag elements for each user-defined interface range as a child of the **<interface-range>** tag element, instead of displaying them as children of the interface elements that are members of the interface range. This display mode parallels the default behavior of the CLI configuration mode **show** command, which displays the **[edit groups]** and **[edit interfaces interface-range]** hierarchies as separate hierarchies in the configuration.

To request that the Junos XML protocol server not display the **<groups>**, **<apply-groups>**, or **<interface-range>** tag elements separately, but instead enclose tag elements inherited from user-defined groups or interface ranges within the inheriting tag elements, a client application includes the **inherit="inherit"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration inherit="inherit"/>

  <!-- OR -->

  <get-configuration inherit="inherit">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To request that the Junos XML protocol server include tag elements that are inherited from the **junos-defaults** group as well as user-defined configuration groups and interface-ranges, the client application includes the **inherit="defaults"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

```
<rpc>
  <get-configuration inherit="defaults"/>

  <!-- OR -->

  <get-configuration inherit="defaults">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Information to Return” on page 89.

When the client includes the **inherit="inherit"** attribute, the output includes the same information as the output from the following CLI configuration mode command, and does not include configuration elements inherited from the **junos-defaults** group:

```
user@host# show | display inheritance | except ##
```

When the client includes the **inherit="defaults"** attribute, the output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance defaults | except ##
```

In both cases, the Junos XML protocol server encloses its output in the **<rpc-reply>** tag element and either the **<configuration>** tag element (for Junos XML-tagged output) or **<configuration-text>** tag element (for formatted ASCII output). For information about the attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 70.

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

  <!-- OR -->

    <configuration-text>
      <!-- formatted ASCII configuration statements -->
    </configuration-text>
</rpc-reply>
```

The **inherit** attribute can be combined with one or more of the following attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in "Requesting a Change Indicator for Configuration Elements" on page 76
- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70
- **format**, which is described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72
- **groups**, which is described in "Displaying the Source Group for Inherited Configuration Elements" on page 82
- **interface-ranges**, which is described in "Displaying the Source Interface Range for Inherited Configuration Elements" on page 87

The application can also include the **inherit** attribute after requesting an indicator for identifiers (as described in "Requesting an Indicator for Identifiers" on page 74).

Displaying the Source Group for Inherited Configuration Elements

To request that the Junos XML protocol server indicate the configuration group from which configuration elements are inherited, a client application combines the **groups="groups"** attribute with the **inherit** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
    <get-configuration inherit="(defaults | inherit)" groups="groups"/>

  <!-- OR -->

    <get-configuration inherit="(defaults | inherit)" groups="groups">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```


For information about the tag elements to enclose in the **<get-configuration>** tag element, see "Specifying the Scope of Configuration Information to Return" on page 89.

When the **groups="groups"** attribute is combined with the **inherit="inherit"** attribute, the output includes the same information as the output from the following CLI configuration mode command, and does not include configuration elements inherited from the **junos-defaults** group:

```
user@host# show | display inheritance | display xml groups
```

When the **groups="groups"** attribute is combined with the **inherit="defaults"** attribute, the output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance defaults
```

The **inherit** and **groups** attributes can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in "Requesting a Change Indicator for Configuration Elements" on page 76.
- **database**, which is described in "Requesting Information from the Committed or Candidate Configuration" on page 70.
- **format**, which is described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72. The application can request either Junos XML-tagged or formatted ASCII output:
  - If the output is tagged with Junos XML tag elements (the **format="xml"** attribute is included or the **format** attribute is omitted), the Junos XML protocol server includes the **junos:group="*source-group*"** attribute in the opening tags of configuration elements that are inherited from configuration groups and encloses its response in **<configuration>** 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 70.

    ```
    <rpc-reply xmlns:junos="URL">
      <configuration attributes>
      <!-- For each inherited element -->
        <!-- opening-tags-for-parents-of-the-element -->
          <inherited-element junos:group="source-group">
            <inherited-child-of-inherited-element junos:group="source-group">
              <!-- inherited-children-of-child junos:group="source-group" -->
            </inherited-child-of-inherited-element>
          </inherited-element>
        <!-- closing-tags-for-parents-of-the-element -->
      </configuration>
    </rpc-reply>
    ```

  - If the output is formatted ASCII text (the **format="text"** attribute is included), the Junos XML protocol server encloses its response in **<configuration-text>** and **<rpc-reply>** tag elements, and inserts three commented lines immediately above each inherited element, as in the following:

    ```
    <rpc-reply xmlns:junos="URL">
      <configuration-text>
    */ For each inherited element */
    ```

```
        /* parent levels for the element */
          ##
          ## 'inherited-element' was inherited from group 'source-group'
          ##
          inherited-element {
            ##
            ## 'inherited-child' was inherited from group 'source-group'
            ##
            inherited-child {
              ... child statements of inherited-child ...
            }
          }
        /* closing braces for parent levels for the element */
      </configuration-text>
    </rpc-reply>
```

- **interface-ranges**, which is described in "Displaying the Source Interface Range for Inherited Configuration Elements" on page 87

The application can also include the **inherit** and **groups** attributes after requesting an indicator for identifiers (as described in "Requesting an Indicator for Identifiers" on page 74).

Examples: Specifying Output Format for Configuration Groups

The following sample configuration hierarchy defines a configuration group called **interface-group**. The **apply-groups** statement applies the statements in the group at the **[edit interfaces]** hierarchy level:

```
[edit]
groups {
  interface-group {
    interfaces {
      so-1/1/1 {
        encapsulation ppp;
      }
    }
  }
}
apply-groups interface-group;
interfaces {
  fxp0 {
    unit 0 {
      family inet {
        address 192.168.4.207/24;
      }
    }
  }
}
```

When the **inherit** attribute is not included in the **<get-configuration/>** tag, the output includes the **<groups>** and **<apply-groups>** tag elements as separate items. The **<groups>** tag element encloses the tag elements defined in the **interface-group** configuration group. The placement of the **<apply-groups>** tag element directly above the **<interfaces>** tag

element indicates that the **[edit interfaces]** hierarchy inherits the statements defined in the **interface-group** configuration group.

Client Application

```
<rpc>
  <get-configuration/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
      junos:changed-localtime="timestamp">
    <groups>
      <name>interface-group</name>
      <interfaces>
        <interface>
          <name>so-1/1/1</name>
          <encapsulation>ppp</encapsulation>
        </interface>
      </interfaces>
    </groups>
    <apply-groups>interface-group</apply-groups>
    <interfaces>
      <interface>
        <name>fxp0</name>
        <unit>
          <name>0</name>
          <family>
            <inet>
              <address>
                <name>192.168.4.207/24</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>
```

T1188

When the **inherit** attribute is included in the **<get-configuration/>** tag, the **<interfaces>** tag element encloses the tag elements defined in the **interface-group** configuration group. The **<groups>** and **<apply-groups>** tag elements are not displayed.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration inherit="inherit"/>
</rpc>
                                <rpc-reply xmlns:junos="URL">
                                  <configuration junos:changed-seconds="seconds" \
                                       junos:changed-localtime="timestamp">
                                    <interfaces>
                                      <interface>
                                        <name>fxp0</name>
                                        <unit>
                                          <name>0</name>
                                          <family>
                                            <inet>
                                              <address>
                                                <name>192.168.4.207/24</name>
                                              </address>
                                            </inet>
                                          </family>
                                        </unit>
                                      </interface>
                                      <interface>
                                        <name>so-1/1/1</name>
                                        <encapsulation>ppp</encapsulation>
                                      </interface>
                                    </interfaces>
                                  </configuration>
                                </rpc-reply>
```

T1189

When the **groups="groups"** attribute is combined with the **inherit** attribute in the **<get-configuration/>** tag, the **<interfaces>** tag element encloses the tag elements defined in the **interface-group** configuration group, which are marked with the **junos:group="interface-group"** attribute.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration inherit="inherit" groups="groups"/>
</rpc>
                              <rpc-reply xmlns:junos="URL">
                                <configuration junos:changed-seconds="seconds" \
                                     junos:changed-localtime="timestamp">
                                  <interfaces>
                                    <interface>
                                      <name>fxp0</name>
                                      <unit>
                                        <name>0</name>
                                        <family>
                                          <inet>
                                            <address>
                                              <name>192.168.4.207/24</name>
                                            </address>
                                          </inet>
                                        </family>
                                      </unit>
                                    </interface>
                                    <interface junos:group="interface-group">
                                      <name junos:group="interface-group">so-1/1/1</name>
                                      <encapsulation junos:group="interface-group">ppp</encapsulation>
                                    </interface>
                                  </interfaces>
                                </configuration>
                              </rpc-reply>
```

T1190

Displaying the Source Interface Range for Inherited Configuration Elements

To request that the Junos XML protocol server indicate the interface range from which configuration elements are inherited, a client application combines the **interface-ranges="interface-ranges"** attribute with the **inherit** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
    <get-configuration inherit="inherit" interface-ranges="interface-ranges"/>

  <!-- OR -->

    <get-configuration inherit="inherit" interface-ranges="interface-ranges">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Information to Return” on page 89.

When the **interface-ranges="interface-ranges"** attribute is combined with the **inherit="inherit"** attribute, the output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance | display xml interface-ranges
```

The **inherit** and **interface-ranges** attributes can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in “Requesting a Change Indicator for Configuration Elements” on page 76.
- **database**, which is described in “Requesting Information from the Committed or Candidate Configuration” on page 70.
- **format**, which is described in “Requesting Output as Formatted ASCII Text or Junos XML Tag Elements” on page 72. The application can request either Junos XML-tagged or formatted ASCII output:
  - If the output is tagged with Junos XML tag elements (the **format="xml"** attribute is included or the **format** attribute is omitted), the Junos XML protocol server includes the **junos:interface-range="*source-interface-range*"** attribute in the opening tags of configuration elements that are inherited from an interface range and encloses its response in **<configuration>** 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 70.

    ```
    <rpc-reply xmlns:junos="URL">
      <configuration attributes>
        <interfaces>
          <!-- For each inherited element -->
          <interface junos:interface-range="source-interface-range">
            <inherited-element junos:interface-range="source-interface-range">
              <inherited-child-of-inherited-element
                    junos:interface-range="source-interface-range">
    ```

```
            <!-- inherited-children-of-child
                 junos:interface-range="source-interface-range" -->
          </inherited-child-of-inherited-element>
        </inherited-element>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>
```

- If the output is formatted ASCII text (the **format="text"** attribute is included), the Junos XML protocol server encloses its response in **<configuration-text>** and **<rpc-reply>** tag elements:

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
    interfaces {
      <!-- For each inherited element -->
      interface-name {
        inherited-element {
          inherited-child {
            ... child statements of inherited-child ...
          }
        }
      }
    }
  </configuration-text>
</rpc-reply>
```

- **groups**, which is described in "Displaying the Source Group for Inherited Configuration Elements" on page 82

The application can also include the **inherit** and **interface-ranges** attributes after requesting an indicator for identifiers (as described in "Requesting an Indicator for Identifiers" on page 74).

Comparing Configuration Changes with a Prior Version

In the CLI, when you want to compare the active or candidate configuration to a previously committed configuration, you use the **compare** command. In operational mode, you compare the active configuration to a prior version using the **show configuration | compare rollback *rollback-number*** command. In configuration mode, you compare the candidate configuration to a previously committed configuration using the **show | compare rollback *rollback-number*** command.

The **compare rollback *rollback-number*** command compares the selected configuration with a previously committed configuration and displays the differences between the two. The ***rollback-number*** for the most recently saved configuration is 0, and the oldest saved configuration is 49.

To request that the Junos XML protocol server display the differences between the active or candidate configuration and a previously committed configuration, a client application includes the **compare** and **rollback** attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
```

```
    <get-configuration compare="rollback" rollback="[0-49]" format="text"/>

  <!-- OR -->

    <get-configuration compare="rollback" rollback="[0-49]" format="text">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
</rpc>
```

The client application can include the **database** attribute to specify whether to compare the active or candidate configuration to the previously committed configuration. If the **database** attribute is omitted, the candidate configuration is used. If the **rollback=*"rollback-number"*** attribute is not included, rollback configuration number 0 is used for comparison.

By default, the **<get-configuration>** operation returns Junos XML-tagged output. However, when the **compare** attribute is included, the **<get-configuration>** operation returns the output formatted as ASCII text even if the **format="text"** attribute is not present. If the client application attempts to include the **format="xml"** attribute when the **compare="rollback"** attribute is present, the protocol server will return an **<xnm:error>** element indicating an error.

The comparison output is enclosed in the **<configuration-information>** and **<configuration-output>** tags. The output uses the following conventions to specify the differences between configurations:

- Statements that are only in the active or candidate configuration are prefixed with a plus sign (+).
- Statements that are only in the comparison file are prefixed with a minus sign (–).
- Statements that are unchanged are prefixed with a single blank space ( ).

Specifying the Scope of Configuration Information to Return

By including the appropriate optional child tag elements in the **<get-configuration>** tag element, a client application can request the entire configuration or specific portions of the configuration, as described in the following sections:

- Requesting the Complete Configuration on page 90
- Requesting a Hierarchy Level or Container Object Without an Identifier on page 91
- Requesting All Configuration Objects of a Specified Type on page 92
- Requesting a Specified Number of Configuration Objects on page 93
- Requesting Identifiers Only on page 95
- Requesting One Configuration Object on page 97
- Requesting a Subset of Objects by Using Regular Expressions on page 99
- Requesting Multiple Configuration Elements Simultaneously on page 102

Requesting the Complete Configuration

To request the entire configuration, a client application encloses the **<get-configuration/>** tag in an **<rpc>** tag element:

```
<rpc>
  <get-configuration/>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested configuration in **<configuration>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- Junos XML tag elements for all configuration elements -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the **<get-configuration/>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The following example shows how to request the complete candidate configuration tagged with Junos XML tag elements (the default). 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-configuration/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration 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>
</rpc-reply>
```

T11191

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 **<get-configuration>** 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 level's immediate parent level. An empty tag represents the requested level. The entire request is enclosed in an **<rpc>** tag element.

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
        <requested-level/>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested section of the configuration in **<configuration>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the level -->
      <hierarchy-level>
        <!-- child tag elements of the level -->
      </hierarchy-level>
    <!-- closing tags for each parent of the level -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

The following example shows how to request the contents of the **[edit system login]** hierarchy level in the candidate configuration. The output is tagged with Junos XML tag elements (the default).

Client Application	Junos XML Protocol Server

```
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
      junos:changed-localtime="timestamp">
    <system>
      <login>
        <user>
          <name>barbara</name>
          <full-name>Barbara Anderson</full-name>
          <!- -other child tags for this user- ->
        </user>
        <!- -other children of <login>- ->
      </login>
    </system>
  </configuration>
</rpc-reply>
```

T1192

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 **<get-configuration>** 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. An empty tag represents the requested object type. The entire request is enclosed in an **<rpc>** tag element.

This type of request is useful when the object's parent hierarchy level has child objects of multiple types and the application is requesting just one of the types. If the requested object is the only possible child type, then this type of request yields the same output as a request for the complete parent hierarchy (described in "Requesting a Hierarchy Level or Container Object Without an Identifier" on page 91).

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
        <object-type/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in **<configuration>** 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 70.

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

```
<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>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

Requesting a Specified Number of Configuration Objects

To request information about a specific number of configuration objects of a specific type, a client application emits the **<get-configuration>** tag element and encloses the tag elements that represent 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. An empty tag represents the requested object type, and the following attributes are included in it:

- **count**, to specify the number of objects to return
- **start**, to specify the index number of the first object to return (**1** for the first object, **2** for the second, and so on)

(If the application is requesting only the first object in the hierarchy, it includes the **count="1"** attribute and omits the **start** attribute.) The application encloses the entire request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
        <object-type  count="count" start="index"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in **<configuration>** and **<rpc-reply>** tag elements, starting with the object specified by the **start** attribute and running consecutively. In the opening container tag for each object, it includes two attributes:

- **junos:position**, to specify the object's numerical index
- **junos:total**, to report the total number of such objects that exist in the hierarchy

In the following, 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 70.)

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object type -->
      <first-object junos:position="index1" junos:total="total">
        <name>identifier-for-first-object</name>
        <!-- other child tag elements of the first object -->
      </first-object>
      <second-object junos:position="index2" junos:total="total">
        <name>identifier-for-second-object</name>
        <!-- other child tag elements of the second object -->
      </second-object>
      <!-- additional objects -->
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

The **junos:position** and **junos:total** attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the **<get-configuration>** tag element (as described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72).

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

The following example shows how to request the third and fourth Junos user accounts at the **[edit system login]** hierarchy level. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login>
          <user count="2" start="3"/>
        </login>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
      junos:changed-localtime="timestamp">
    <system>
      <login>
        <user junos:position="3" junos:total="22">
          <name>barbara</name>
          <uid>1423</uid>
          <class>operator</class>
        </user>
        <user junos:position="4" junos:total="22">
          <name>carlo</name>
          <uid>1426</uid>
          <class>operator</class>
        </user>
      </login>
    </system>
  </configuration>
</rpc-reply>
```

T1193

Requesting Identifiers Only

To request just the identifier tag element for configuration objects of a specified type in a hierarchy, a client application emits a **<get-configuration>** 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. An empty tag represents the requested object type, and the **recurse="false"** attribute is included. The entire request is enclosed in an **<rpc>** tag element.

To request the identifier for all objects of a specified type, the client application includes only the recurse="false" attribute:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
        <object-type recurse="false"/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

To request the identifier for a specified number of objects, the client application combines the **recurse="false"** attribute with the **count** and **start** attributes discussed in "Requesting a Specified Number of Configuration Objects" on page 93:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
        <object-type recurse="false" count="count" start="index"/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in **<configuration>** and **<rpc-reply>** tag elements. If the application has requested a specified number of objects, the **junos:position** and **junos:total** attributes are included in the opening tag for each object, as described in "Requesting a Specified Number of Configuration Objects" on page 93. In the following, 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 70.)

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object type -->
      <first-object [junos:position="index1" junos:total="total"]>
        <name>identifier-for-first-object</name>
      </first-object>
      <second-object [junos:position="index2" junos:total="total"]>
        <name>identifier-for-second-object</name>
      </second-object>
      <!-- additional instances of the object -->
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

The **junos:position** and **junos:total** attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the **<get-configuration>** tag element (as described in "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72).

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

The following example shows how to request the identifier for each interface configured at the **[edit interfaces]** hierarchy level. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration>
    <configuration>
      <interfaces>
        <interface recurse="false"/>
      </interfaces>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
      junos:changed-localtime="timestamp">
    <interfaces>
      <interface>
          <name>fe-0/0/0</name>
      </interface>
      <interface>
          <name>fxp0</name>
      </interface>
      <interface>
          <name>lo0</name>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>
```

T1194

Requesting One Configuration Object

To request information about a single configuration object, a client application emits the **<get-configuration>** tag element and encloses the tag elements that represent the entire statement path down to the object, starting with the **<configuration>** tag element. To represent the requested object, the application emits only the container tag element and identifier tag elements (each complete with the identifier value) for the object. It does not emit tag elements that represent other object characteristics. It encloses the entire request in an **<rpc>** tag element. In the following, the identifier tag element is called **name**:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
        <object>
          <name>identifier</name>
        </object>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

When the client application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested object in **<configuration>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object -->
      <object>
        <!-- child tag elements of the object -->
      </object>
    <!-- closing tags for each parent of the object -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

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. To specify the desired object, the client application emits the **<name>local</name>** identifier tag element as the innermost tag element. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration>
    <configuration>
      <routing-options>
        <multicast>
          <scope>
            <name>local</name>
          </scope>
        </multicast>
      </routing-options>
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration 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>
</rpc-reply>
```

T11195

Requesting a Subset of Objects by Using Regular Expressions

To request information about only those instances of a configuration object type that have a specified set of characters in their identifier names, a client application includes the **matching** attribute with a regular expression that matches the identifier name. For example, the application can request information about just the SONET/SDH interfaces at the **[edit interfaces]** hierarchy level by specifying the characters **so-** at the start of the regular expression.

Using the **matching** attribute enables the application to represent the objects to return in a form similar to the XML Path Language (XPath) representation, which is described in *XML Path Language (XPath) Version 1.0*, available from the World Wide Web Consortium (W3C) at http://www.w3.org/TR/xpath . In an XPath representation, an object and its parent levels are an ordered series of tag element names separated by forward slashes. The angle brackets around tag element names are omitted, and the opening tag is used to represent the entire tag element. For example, the following XPath:

```
configuration/system/radius-server/name
```

is equivalent to the following tagged representation:

```
<configuration>
  <system>
    <radius-server>
      <name/>
    </radius-server>
  </system>
</configuration>
```

The application includes the **matching** attribute in the empty tag that represents a parent level for the object type. As with all requests for configuration information, the client emits a **<get-configuration>** 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 level at which the **matching** attribute is included. The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
        <level  matching="matching-expression"/>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </get-configuration>
</rpc>
```

In the value for the **matching** attribute, each level in the XPath-like representation can be either a full level name or a regular expression that matches the identifier name of one or more instances of an object type:

object-type[name='*regular-expression*']"

The regular expression uses the notation defined in POSIX Standard 1003.2 for extended (modern) UNIX regular expressions. Explaining regular expression syntax is beyond the scope of this document, but Table 7 on page 100 specifies which character or characters are matched by some of the regular expression operators that can be used in the expression. In the descriptions, the term *term* refers to either a single alphanumeric character or a set of characters enclosed in square brackets, parentheses, or braces.

NOTE: The matching attribute is not case-sensitive.

Table 7: Regular Expression Operators for the matching Attribute

Operator	Matches
. (period)	One instance of any character except the space.
***** (asterisk)	Zero or more instances of the immediately preceding term.
+ (plus sign)	One or more instances of the immediately preceding term.
? (question mark)	Zero or one instance of the immediately preceding term.
| (pipe)	One of the terms that appear on either side of the pipe operator.
^ (caret)	The start of a line, when the caret appears outside square brackets. One instance of any character that does not follow it within square brackets, when the caret is the first character inside square brackets.

Table 7: Regular Expression Operators for the matching Attribute *(continued)*

Operator	Matches
$ (dollar sign)	The end of a line.
[ ] (paired square brackets)	One instance of one of the enclosed alphanumeric characters. To indicate a range of characters, use a hyphen (-) to separate the beginning and ending characters of the range. For example, **[a-z0-9]** matches any letter or number.
( ) (paired parentheses)	One instance of the evaluated value of the enclosed term. Parentheses are used to indicate the order of evaluation in the regular expression.

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested object in **<configuration>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the parent level -->
      <parent-level>
        <first-matching-object>
          <!-- child tag elements for the first object -->
        </first-matching-object>
        <second-matching-object>
          <!-- child tag elements for the second object -->
        </second-matching-object>
        <!-- additional instances of the object -->
      </parent-level>
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

The application can combine one or more of the **count**, **start**, and **recurse** attributes along with the **matching** attribute, to limit the set of possible matches to a specific range of objects, to request only identifiers, or both. For more information about those attributes, see "Requesting a Specified Number of Configuration Objects" on page 93 and "Requesting Identifiers Only" on page 95.

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII text or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see "Specifying the Source and Output Format of Configuration Information" on page 69.

The application can request additional configuration elements of the same or other types in the same **<get-configuration>** tag element by including the appropriate tag elements. For more information, see "Requesting Multiple Configuration Elements Simultaneously" on page 102.

The following example shows how to request just the identifier for the first two SONET/SDH interfaces configured at the **[edit interfaces]** hierarchy level.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-configuration>
    <configuration>
      <interfaces matching="interface[name='so-.*']" count="2" recurse="false">
    </configuration>
  </get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
          junos:changed-localtime="timestamp">
    <interfaces>
      <interface junos:position="41" junos:total="65">
        <name>so-0/0/0</name>
      </interface>
      <interface junos:position="42" junos:total="65">
        <name>so-0/0/1</name>
      </interface>
    <interfaces>
  </configuration>
</rpc-reply>
```

T1196

Requesting Multiple Configuration Elements Simultaneously

Within a **<get-configuration>** tag element, a client application can request multiple configuration elements of the same type or different types. The request includes only one **<configuration>** tag element (the Junos XML protocol server returns an error if there is more than one).

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. As an 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 parent tag element -->
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <services/>
        <radius-server>
          <name/>
        </radius-server>
      </system>
    </configuration>
  </get-configuration>
</rpc>

<!-- separate parent tag element for each request -->
<rpc>
  <get-configuration>
    <configuration>
```



```
        <system>
          <services/>
        </system>
        <system>
          <radius-server>
            <name/>
          </radius-server>
        </system>
      </configuration>
    </get-configuration>
  </rpc>
```

The client can combine requests for any of the types of information discussed in the following sections:

- Requesting a Hierarchy Level or Container Object Without an Identifier on page 91
- Requesting All Configuration Objects of a Specified Type on page 92
- Requesting a Specified Number of Configuration Objects on page 93
- Requesting Identifiers Only on page 95
- Requesting One Configuration Object on page 97
- Requesting a Subset of Objects by Using Regular Expressions on page 99

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 Junos XML protocol server encloses the XML schema in **<rpc-reply>** and **<xsd:schema>** tag elements:

```
<rpc-reply xmlns:junos="URL">
  <xsd:schema>
    <!-- tag elements for the Junos 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, 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-configuration>** tag element instead, as described in "Requesting Configuration Information" on page 68.

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 104
- Example: Requesting an XML Schema on page 105

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 Junos XML protocol 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 Junos XML protocol server's response appear on multiple lines for legibility only. The Junos XML protocol 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 **Junos XML Protocol Server**

```
<rpc>
  <get-xnm-information>
    <type>xml-schema</type>
    <namespace>junos-configuration</namespace>
  </get-xnm-information>
</rpc>
                                <rpc-reply 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>
```

T1177

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 **Junos XML Protocol 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> - ->
                </xsd:choice >
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <!- - definitions for other hierarchy levels - ->
        </xsd:choice >
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</rpc-reply>
```

T1178

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 Junos XML protocol server encloses its response in **<rpc-reply>**, **<rollback-information>**, and **<configuration>** tag elements. The **<load-success/>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <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 Junos XML protocol server encloses its response in **<rpc-reply>**, **<rollback-information>**, **<configuration-information>**, and **<configuration-output>** tag elements. For more information about how ASCII output is formatted, see “Requesting Output as Formatted ASCII Text or Junos XML Tag Elements” on page 72.

```
  <rpc-reply xmlns:junos="URL">
    <rollback-information>
      <load-success/>
      <configuration-information>
        <configuration-output>
          <!-- formatted ASCII text for 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>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration 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>
```

T1197

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 Junos XML protocol server encloses its response in **<rpc-reply>**, **<rollback-information>**, **<configuration-information>**, and **<configuration-output>** tag elements. The **<load-success/>** tag is a side effect of the implementation and does not affect the results.

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration-information>
      <configuration-output>
        <!-- formatted ASCII text representing the changes -->
      </configuration-output>
    </configuration-information>
  </rollback-information>
</rpc-reply>
```

The information in the **<configuration-output>** tag element is formatted ASCII text 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.

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>
</r pc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <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>
</r pc-reply>
```

T1170

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 Junos XML protocol server encloses its response in **<rpc-reply>**, **<rescue-information>**, and **<configuration>** tag elements. The **<load-success/>** 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 70.

```
<rpc-reply xmlns:junos="URL">
  <rescue-information>
    <load-success/>
    <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 Junos XML protocol server encloses its response in **<rpc-reply>**, **<rescue-information>**, **<configuration-information>**, and **<configuration-output>** tag elements. For more information about how ASCII output is formatted, see "Requesting Output as Formatted ASCII Text or Junos XML Tag Elements" on page 72.

```
<rpc-reply xmlns:junos="URL">
  <rescue-information>
    <load-success/>
    <configuration-information>
      <configuration-output>
        <!-- formatted ASCII text representing the rescue configuration -->
      </configuration-output>
    </configuration-information>
  </rescue-information>
</rpc-reply>
```


CHAPTER 5

Changing Configuration Information

This chapter explains how to use the Junos 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 configuration. The Junos XML protocol **<load-configuration>** tag element and its attributes correspond to configuration mode commands in the Junos OS CLI, which 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 the following topics:

Overview of Configuration Changes

To change configuration information, the client application performs the procedures described in the indicated sections:

1. Establishes a connection to the Junos XML protocol server on the routing, switching, or security platform, as described in "Connecting to the Junos XML Protocol Server" on page 39.
2. Opens a Junos XML protocol session, as described in "Starting the Junos XML Protocol Session" on page 40.
3. (Optional) Locks the candidate configuration or creates a private copy, as described in "Locking the Candidate Configuration" on page 55 and "Creating a Private Copy of the Configuration" on page 58. Locking the configuration prevents other users or applications from changing it at the same time. Creating a private copy enables the

application to make changes without affecting the candidate or active configuration until the copy is committed.

4. Encloses the **<load-configuration>** tag element in an **<rpc>** tag element. By including various attributes in the **<load-configuration/>** tag or opening **<load-configuration>** tag, the application can provide the configuration data either in a file or as a directly loaded tag stream, and either as Junos XML tag elements or formatted ASCII text. It can completely replace the existing configuration or can specify the manner in which the Junos XML protocol server loads the data into the existing candidate or copy. The basic syntax is as follows:

```
<rpc>
  <!-- If providing configuration data in a file -->
    <load-configuration url="file" [optional attributes]

  <!-- If providing configuration data in a data stream -->
    <load-configuration [optional attributes]>
      <!-- configuration data -->
    </load-configuration>
</rpc>
```

5. Accepts the tag stream emitted by the Junos XML protocol server in response to each request and extracts its content, as described in “Parsing the Junos XML Protocol Server Response” on page 51.

   The Junos XML protocol server confirms that it incorporated the configuration data by returning the **<load-configuration-results>** tag element and **<load-success/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

   If the load operation fails, the **<load-configuration-results>** tag element instead encloses the **<load-error-count>** tag element, which indicates the number of errors that occurred. In this case, the application or an administrator must eliminate the errors before committing the configuration.

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-error-count>count</load-error-count>
  </load-configuration-results>
</rpc-reply>
```

6. (Optional) Verifies the syntactic correctness of a configuration before attempting to commit it, as described in “Verifying a Configuration Before Committing It” on page 141.

7. Commits changes made to the configuration, as described in “Committing a Configuration” on page 141.

8. Unlocks the candidate configuration if it is locked, as described in "Unlocking the Candidate Configuration" on page 56. Other users and applications cannot change the configuration while it remains locked.
9. Ends the Junos XML protocol session and closes the connection to the device, as described in "Ending a Junos XML Protocol Session and Closing the Connection" on page 58.

Specifying the Source and Format of New Configuration Data

A client application can provide new configuration data either in a file or as a data stream, and as either Junos XML tag elements or formatted ASCII text. See the following sections:

- Providing Configuration Data in a File on page 113
- Providing Configuration Data as a Data Stream on page 114
- Defining Configuration Data as Formatted ASCII Text or Junos XML Tag Elements on page 115

Providing Configuration Data in a File

To provide new configuration data in a file, a client application encloses the **<load-configuration/>** tag with the **url** attribute in an **<rpc>** tag element. If the data is Junos XML tag elements, it either includes the **format="xml"** attribute or omits the **format** attribute:

```
<rpc>
  <load-configuration url="file-location"/>
</rpc>
```

If the data is formatted ASCII text, the **format="text"** attribute is included:

```
<rpc>
  <load-configuration url="file-location" format="text"/>
</rpc>
```

Before loading the file, the client application or an administrator saves either Junos XML tag elements (enclosed in a **<configuration>** tag element) or formatted ASCII as the contents of the file (formatted ASCII text is not enclosed in a **<configuration-text>** tag element in the file). For information about the syntax for the data in the file, see "Defining Configuration Data as Formatted ASCII Text or Junos XML Tag Elements" on page 115.

The value of the **url** attribute can be a local file path, an FTP location, or a Hypertext Transfer Protocol (HTTP) URL:

- A local filename can have one of the following forms:
  - ***/path/filename***—File on a mounted file system, either on the local flash disk or on hard disk.
  - **a:*filename*** or **a:*path/filename***—File on the local drive. The default path is **/** (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.
- A filename on an FTP server has the following form:

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

- A filename on an HTTP server has the following form:

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

In each case, the default value for the ***path*** variable is the home directory for the username. To specify an absolute path, the application starts the path with the characters **%2F**; for example, **ftp://*username*:*password*@*hostname*/%2F*path*/*filename***.

The **url** attribute can be combined with one or more of the following other attributes in the **<load-configuration/>** tag:

- **format**, which is described in "Defining Configuration Data as Formatted ASCII Text or Junos XML Tag Elements" on page 115.
- **action**, which is described in "Replacing the Entire Configuration" on page 116 and the subsections of "Creating, Modifying, or Deleting Configuration Elements" on page 117.

The following example shows how to incorporate Junos XML-tagged configuration data stored in the file **/var/configs/user-accounts** on the FTP server called **cfg-server.mycompany.com**. The opening **<load-configuration>** tag appears on two lines for legibility only.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration \
    url="ftp://admin:AdminPwd@cfg-server.mycompany.com/var/configs/user-accounts"/>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1179

Providing Configuration Data as a Data Stream

To provide new configuration data as a data stream, a client application encloses the **<load-configuration>** tag element in an **<rpc>** tag element.

To define the configuration elements to change as Junos XML tag elements, the application emits 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:

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- tag elements representing the configuration data -->
    </configuration>
  </load-configuration>
</rpc>
```

To define the configuration elements to change as formatted ASCII text, the application encloses them in a **<configuration-text>** tag element and includes the **format="text"** attribute in the opening **<load-configuration>** tag:

```
<rpc>
```



```
<load-configuration format="text">
  <configuration-text>
    /* formatted ASCII configuration data */
  </configuration-text>
</load-configuration>
</rpc>
```

For information about the syntax for Junos XML tag elements and formatted ASCII text, see "Defining Configuration Data as Formatted ASCII Text or Junos XML Tag Elements" on page 115.

Defining Configuration Data as Formatted ASCII Text or Junos XML Tag Elements

As discussed in "Providing Configuration Data in a File" on page 113 and "Providing Configuration Data as a Data Stream" on page 114, a client application can provide new configuration data to the Junos XML protocol server either in a file or as a data stream emitted during the Junos XML protocol session. In both cases, it can use either Junos XML tag elements or formatted ASCII text to define the new configuration data.

If the application uses Junos XML tag elements, it includes the tag elements representing all levels of the configuration hierarchy from the root (the **<configuration>** tag element) down to each new or changed element. The notation is the same as that used to request configuration information, and is described in detail in "Overview of Configuration Changes" on page 111.

```
<configuration>
  <!-- tag elements representing the configuration data -->
</configuration>
```

If the application provides the new data as formatted ASCII text, it uses the standard Junos OS CLI notation to indicate the hierarchical relationships between configuration statements—the newline character, tabs and other white space, braces, and square brackets. For each new or changed element, the complete statement path is specified, starting with the top-level statement that appears directly under the **[edit]** hierarchy level.

When ASCII text is provided as a data stream, it is enclosed in the **<configuration-text>** tag element:

```
<configuration-text>
  /* formatted ASCII configuration statements */
</configuration-text>
```

When ASCII text is provided in a previously saved file, the **<configuration-text>** tag element is not included in the file.

When providing new data as ASCII text, the application also includes the **format="text"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag.

```
<rpc>
  <load-configuration url="file-location" format="text"/>
</rpc>
```

```
<rpc>
  <load-configuration format="text">
    <configuration-text>
```

```
      /* formatted ASCII configuration data */
    </configuration-text>
  </load-configuration>
</rpc>
```

The **format** attribute can be combined with one or more of the following attributes:

- **url**, which is discussed in "Providing Configuration Data in a File" on page 113.
- **action**, which is discussed in "Replacing the Configuration with New Data" on page 116 and the subsections of "Creating, Modifying, or Deleting Configuration Elements" on page 117.

For reference pages for the **<configuration>** and **<configuration-text>** tag elements, see the *Junos XML API Operational Reference*.

Replacing the Entire Configuration

A client application can completely replace the current candidate configuration or a private copy of it, either with new data or by rolling back to a previous configuration. See the following sections:

- Replacing the Configuration with New Data on page 116
- Rolling Back to a Previous or Rescue Configuration on page 117

For instructions about modifying individual configuration elements, see "Creating, Modifying, or Deleting Configuration Elements" on page 117.

Replacing the Configuration with New Data

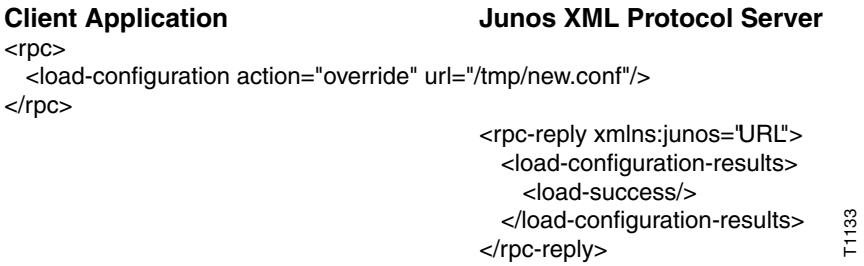
To discard the entire candidate configuration or private copy and replace it with new configuration data, a client application includes the **action="override"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag:

```
<rpc>
  <!-- For a file -->
    <load-configuration action="override" url="file" [format="text"]/>

  <!-- For a data stream -->
    <load-configuration action="override" [format="text"]>
      <!-- configuration data -->
    </load-configuration>
</rpc>
```

For more information about the **url** and **format** attributes and the syntax for the new configuration data, see "Specifying the Source and Format of New Configuration Data" on page 113.

The following example shows how to specify that the contents of the file **/tmp/new.conf** replace the entire candidate configuration. The file contains Junos XML tag elements (the default), so the **format** attribute is omitted.



Rolling Back to a Previous or Rescue Configuration

The routing, switching, or security platform stores a copy of the most recently committed configuration and up to 49 additional previous configurations. To replace the candidate configuration or private copy with a previously committed configuration, a client application includes the **rollback="*index*"** attribute in the **<load-configuration/>** tag, where ***index*** is the numerical index of the appropriate previous configuration. The index for the most recently committed configuration is **0** (zero), and the index for the oldest possible previous configuration is **49**.

```
<rpc>
  <load-configuration rollback="index">
</rpc>
```

To replace the configuration with the rescue configuration, include the **rescue="rescue"** attribute in the **<load-configuration/>** tag.

```
<rpc>
  <load-configuration rescue="rescue"/>
</rpc>
```

For more information about rollback and rescue configurations, see the *Junos OS CLI User Guide*.

Creating, Modifying, or Deleting Configuration Elements

In addition to replacing the entire configuration (as described in "Replacing the Entire Configuration" on page 116), a client application can create, modify, or delete one or more configuration elements (hierarchy levels and configuration objects) in the candidate configuration or a private copy.

To use Junos XML tag elements to represent an element, the application includes the tag elements representing all levels in the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the element's container tag element. Which attributes and child tag elements are included depends on the operation being performed on the element. The syntax applies both to the contents of a file and to a data stream. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag [operation-attribute="value"]>
      <name>identifier</name> <!-- if the element has an identifier -->
      <!-- other child tag elements --> <!-- if appropriate for the operation -->
</container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent an element, the application includes the complete statement path, starting with a statement that can appear directly under the **[edit]** hierarchy level. The attributes and child statements to include depend on the operation being performed on the element. The set of statements is enclosed in a **<configuration-text>** tag element when the application provides a data stream. When saving statements to a file for later loading, omit the **<configuration-text>** tag element.

```
<configuration-text>
/* statements for parent levels of the element */
  operation-to-perform: # if appropriate
  element identifier  { # if the element has an identifier
    /* child statements */  # if appropriate for the operation
  }
/* closing braces for parent levels for the element */
</configuration-text>
```

When loading formatted ASCII text, the application includes the **format="text"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag.

For more information about the source and formatting for configuration elements, see "Specifying the Source and Format of New Configuration Data" on page 113.

For information about the operations a client application can perform on configuration elements, see the following sections:

Merging Configuration Elements

By default, the Junos XML protocol server *merges* loaded configuration data into the candidate configuration according to the following rules. (The rules also apply to a private copy of the configuration, but for simplicity the following discussion refers to the candidate configuration only.)

- A configuration element (hierarchy level or configuration object) that exists in the candidate but not in the loaded configuration remains unchanged.
- A configuration element that exists in the loaded configuration but not in the candidate is added to the candidate.
- If a configuration element exists in both configurations, the semantics are as follows:
  - If a child statement of the configuration element (represented by a child tag element) exists in the candidate but not in the loaded configuration, it remains unchanged.
  - If a child statement exists in the loaded configuration but not in the candidate, it is added to the candidate.
  - If a child statement exists in both configurations, the value in the loaded configuration replaces the value in the candidate.

Merge mode is the default mode for new configuration elements, so the application simply emits the **<load-configuration>** tag element in an **<rpc>** tag element:

```
<rpc>
  <!-- For a file -->
    <load-configuration url="file" [format="text"]/>

<!-- For a data stream -->
  <load-configuration [format="text"]>
    <!-- configuration data -->
  </load-configuration>
</rpc>
```

For more information about the **url** and **format** attributes, see "Specifying the Source and Format of New Configuration Data" on page 113.

To explicitly specify merge mode, the application can include the **action="merge"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag, as shown in the examples at the end of this section.

If using Junos XML tag elements to represent the element to merge into the configuration, the application includes the basic tag elements described in "Creating, Modifying, or Deleting Configuration Elements" on page 117. It does not include any attributes in the element's container tag. If adding or changing the value of a child element, the application includes the tag elements for it. If a child remains unchanged, it does not need to be included in the loaded configuration. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name> <!-- if the element has an identifier -->
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

If using formatted ASCII text, the application includes the statement path described in "Creating, Modifying, or Deleting Configuration Elements" on page 117. It does not include a preceding operator, but does include the element's identifier if it has one. If adding or changing the value of a child element, the application includes the tag elements for it. If a child remains unchanged, it does not need to be included in the loaded configuration.

```
<configuration-text>
/* statements for parent levels of the element */
  element identifier  {
     /* child statements if any */
  }
/* closing braces for parent levels for the element */
</configuration-text>
```

The following example shows how to merge in a new interface called **so-3/0/0** at the **[edit interfaces]** hierarchy level in the candidate configuration. The information is provided as Junos XML tag elements (the default).

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration action="merge">
    <configuration>
      <interfaces>
        <interface>
          <name>so-3/0/0</name>
          <unit>
            <family>
              <inet>
                <address>
                  <name>10.0.0.1/8</name>
                </address>
              </inet>
            </family>
          </unit>
        </interface>
      </interfaces>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1131

The following example shows how to use formatted ASCII text to define the same new interface.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration action="merge" format="text">
    <configuration-text>
      interfaces {
        so-3/0/0 {
          unit 0 {
            family inet {
              address 10.0.0.1/8;
            }
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1132

Replacing Configuration Elements

To replace individual configuration elements (hierarchy levels or configuration objects), a client application emits the **<load-configuration>** tag element with the **action="replace"** attribute in an **<rpc>** tag element:

```
<rpc>
  <!-- For a file -->
    <load-configuration action="replace" url="file" [format="text"]/>

  <!-- For a data stream -->
    <load-configuration action="replace" [format="text"]>
      <!-- configuration data -->
    </load-configuration>
</rpc>
```

For more information about the **url** and **format** attributes, see “Providing Configuration Data in a File” on page 113.

To use Junos XML tag elements to represent the replacement, the application includes the basic tag elements described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. Within the container tag, it includes the same child tag elements as for a new element: each of the replacement’s identifier tag elements (if it has them) and all child tag elements being defined for the replacement element. In the following, the identifier tag element is called **<name>**. The application also includes the **replace= “replace”** attribute in the opening container tag:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag replace="replace">
      <name>identifier</name>
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent the element, the application includes the complete statement path described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. As for a new element, it includes each of the replacement’s identifiers (if it has them) and all child statements (with values if appropriate) that it is defining for the replacement. It places the **replace:** statement above the element’s container statement.

```
<configuration-text>
/* statements for parent levels of the element */
  replace:
  element  identifier {
    /* child statements if any */
  }
/* closing braces for parent levels for the element */
</configuration-text>
```

The following example shows how to grant new permissions for the object named **operator** at the **[edit system login class]** hierarchy level. The information is provided in Junos XML-tagged format (the default).

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration action="replace">
    <configuration>
      <system>
        <login>
          <class replace="replace">
            <name>operator</name>
            <permissions>configure</permissions>
            <permissions>admin-control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1135

The following example shows how to use formatted ASCII text to make the same change.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      system {
        login {
        replace:
          class operator {
            permissions [ configure admin-control ];
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1136

Creating New Configuration Elements

To create new configuration elements (hierarchy levels or configuration objects), a client application includes the basic tag elements or formatted ASCII statements described in “Creating, Modifying, or Deleting Configuration Elements” on page 117.

New elements can be created in either merge mode or replace mode, which are described in “Merging Configuration Elements” on page 118 and “Replacing Configuration Elements” on page 121. In replace mode, the application includes the **action="replace"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag.

To use Junos XML tag elements to represent the element, the application includes each of the replacement's identifier tag elements (if it has them) and all child tag elements being defined for the element. In the following, the identifier tag element is called **<name>**. The application does not need to include any attributes in the opening container tag for the new element:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name>
        <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent the element, the application includes each of the replacement's identifiers (if it has them) and all child statements (with values if appropriate) that it is defining for the element. It does not need to include an operator before the new element:

```
<configuration-text>
/* statements for parent levels of the element */
  element identifier {
    /* child statements if any */
  }
/* closing braces for parent levels for the element */
</configuration-text>
```

Replacing Configuration Elements Only If They Have Changed

To replace configuration elements (hierarchy levels and configuration objects) only if they differ in the loaded configuration and the candidate configuration or private copy, the application emits the **<load-configuration>** tag element with the **action="update"** attribute in an **<rpc>** tag element:

```
<rpc>
  <!-- For a file -->
    <load-configuration action="update" url="file" [format="text"]/>

  <!-- For a data stream -->
    <load-configuration action="update" [format="text"]>
      <!-- configuration data -->
  </load-configuration>
</rpc>
```

For more information about the **url** and **format** attributes, see "Specifying the Source and Format of New Configuration Data" on page 113.

This operation is equivalent to the Junos OS CLI **load update** configuration mode command. The Junos configuration management software compares the two configurations. Each configuration element that is different in the loaded configuration replaces its corresponding element in the existing configuration. Elements that are the same in both configurations, or exist only in the existing configuration, remain unchanged. When the configuration is later committed, only system processes that are affected by the changed configuration elements parse the new configuration.

To represent the replacement elements, the application uses the same syntax as for new elements, as described in “Creating New Configuration Elements” on page 122. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name>
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>

OR

<configuration-text>
/* statements for parent levels of the element */
  element identifier {
    /* child statements if any */
  }
/* closing braces for parent levels for the element */
</configuration-text>
```

The following example shows how to update the candidate configuration with the contents of the file **/tmp/new.conf** (which resides on the device). The file contains a complete configuration represented as Junos XML tag elements (the default), so the **format** attribute is omitted.

Client Application	Junos XML Protocol Server
<rpc> <load-configuration action="update" url="/tmp/new.conf"/> </rpc>	
	<rpc-reply xmlns:junos='URL'> <load-configuration-results> <load-success/> </load-configuration-results> </rpc-reply>

T1134

Deleting Configuration Elements

To delete configuration elements (hierarchy levels or configuration objects) from the candidate configuration or private copy, a client application emits the basic tag elements described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. When using Junos XML tag elements to represent the elements to delete, it includes the **delete="delete"** attribute in the opening tag for each one. When using formatted ASCII text, it precedes each element with the **delete:** operator. The placement of the attribute or operator depends on the type of element being deleted, as described in the following sections:

- Deleting a Hierarchy Level or Container Object on page 125
- Deleting a Configuration Object That Has an Identifier on page 125
- Deleting a Single-Value or Fixed-Form Option from a Configuration Object on page 126
- Deleting Values from a Multivalue Option of a Configuration Object on page 128

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 basic tag elements or configuration statements for its parent levels, as described in “Creating, Modifying, or Deleting Configuration Elements” on page 117.

If using Junos XML tag elements, the application includes the **delete="delete"** attribute in the empty tag that represents the level or container object:

```
<configuration>
  <!-- opening tag for each parent level -->
    <level-or-object delete="delete"/>
  <!-- closing tag for each parent level -->
</configuration>
```

If using formatted ASCII text, the application places the **delete:** statement above the level to be removed, which is followed by a semicolon (even though in the existing configuration it is followed by curly braces that enclose its child statements):

```
<configuration-text>
/* statements for parent levels */
  delete:
  object-or-level;
/* closing braces for parent levels */
</configuration-text>
```

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

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <ospf delete="delete"/>
      </protocols>
    </configuration>
  </load-configuration>
</r pc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
      <load-success/>
  </load-configuration-results>
</r pc-reply>
```

T1137

Deleting a Configuration Object That Has an Identifier

To delete a configuration object that has an identifier, a client application includes the basic tag elements or configuration statements for its parent levels, as described in “Creating, Modifying, or Deleting Configuration Elements” on page 117.

If using Junos XML tag elements, the application includes the **delete="delete"** attribute in the opening tag for the object. In the container tag element for the object, it encloses only the identifier tag element, not tag elements that represent any other characteristics of the object. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object delete="delete">
      <name>identifier</name>
    </object>
  <!-- closing tag 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.

If using formatted ASCII text, the application places the **delete:** statement above the object and its identifier:

```
<configuration-text>
/* statements for parent levels of the object */
  delete:
  object identifier;
/* closing braces for parent levels of the object */
</configuration-text>
```

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>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <user delete="delete">
            <name>barbara</name>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1138

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 basic tag elements or configuration

statements for its parent levels, as described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. (For information about deleting an option that can take multiple values, see “Deleting Values from a Multivalue Option of a Configuration Object” on page 128.)

If using Junos XML tag elements, the application includes the **delete="delete"** attribute in the empty tag for each option. It does not include tag elements for children that are to remain in the configuration. In the following, the identifier tag element for the object is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object>
      <name>identifier</name> <!-- if the object has an identifier -->
      <option1 delete="delete"/>
      <option2 delete="delete"/>
      <!-- tag elements for other options to delete -->
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

If using formatted ASCII text, the application places the **delete:** statement above each option:

```
<configuration-text>
/* statements for parent levels of the object */
  object identifier;
  delete:
    option1;
  delete:
    option2;
/* closing braces for parent levels of the object */
</configuration-text>
```

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

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <forwarding-options>
        <sampling>
          <disable delete="delete"/>
        </sampling>
      </forw arding-options>
    </configuration>
  </load-configuration>
</r pc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</r pc-reply>
```

T1140

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 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 basic tag elements or configuration statements for its parent levels, as described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. If using Junos XML tag elements, the application includes the **delete="delete"** attribute in the opening tag for each value. It does not include tag elements that represent values to be retained. In the following, the identifier tag element for the parent object is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the parent object -->
    <parent-object>
      <name>identifier</name>
      <object delete="delete">value1</object>
      <object delete="delete">value2</object>
    </parent-object>
  <!-- closing tag for each parent of the parent object -->
</configuration>
```

If using formatted ASCII text, the application repeats the parent statement for each value and places the **delete:** statement above each paired statement and value:

```
<configuration-text>
/* statements for parent levels of the parent object */
  parent-object identifier;
  delete:
    object value1;
  delete:
    object value2;
/* closing braces for parent levels of the parent object */
</configuration-text>
```

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

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>user-accounts</name>
            <permissions delete="delete">configure</permissions>
            <permissions delete="delete">control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1139

Reordering Elements in Configuration Objects

For most configuration objects, the order in which the object or its children are created is not significant, because the Junos configuration management software stores and displays configuration objects in predetermined positions in the configuration hierarchy. However, some configuration objects—such as routing policies and firewall filters—consist of elements that must be processed and analyzed sequentially in order to produce the intended routing behavior. When a client application uses the Junos XML management protocol to add a new element to an ordered set, the element is appended to the existing list of elements. The client application can then reorder the elements, if appropriate.

To change the order of configuration elements in an ordered set, a client application includes the tag elements described in “Creating, Modifying, or Deleting Configuration Elements” on page 117. It emits the container tag element that represents the ordered set, and encloses the tag element for each identifier of the configuration element that is moving. In the following, the identifier tag element is called **<name>**. In the opening container tag, it includes the **insert="before"** or **insert="after"** attribute to indicate the new position of the moving element relative to another reference element in the set. To identify the reference element, it includes each of the reference element’s identifiers as an attribute in the opening container tag for the ordered set.

In the following, the elements in the set have one identifier, called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="(before | after)" name="referent-value">
      <name>identifier-for-moving-object</name>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```

In the following, each element in the set has two identifiers. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="(before | after)" identifier1="referent-value" \
            identifier2="referent-value">
      <identifier1>value-for-moving-object</identifier1>
      <identifier2>value-for-moving-object</identifier2>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```

The reordering operation is not available when formatted ASCII text is used to represent the configuration data.

The **insert** attribute can be combined with the **inactive** or **active** attribute to deactivate or reactivate the configuration element as it is reordered. For more information, see "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135.

The following example shows how to move a firewall filter called **older-filter**, defined at the **[edit firewall filter]** hierarchy level, and place it after another filter called **newer-filter**. This operation is equivalent to the following configuration mode command:

```
[edit firewall family inet]
user@host# insert filter older-filter after filter newer-filter
```

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration>
    <configuration>
      <firewall>
        <family>
          <inet>
            <filter insert="after" name="newer-filter">
              <name>older-filter</name>
            </filter>
          </inet>
        </family>
      </firewall>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1141

The following example shows how to move an OSPF virtual link defined at the **[edit protocols ospf area *area*]** hierarchy level. The link with identifiers **neighbor-id 192.168.0.3** and **transit-area 1.1.1.1** moves before the link with identifiers **neighbor-id 192.168.0.5** and **transit-area 1.1.1.2**. This operation is equivalent to the following configuration mode command (which appears on two lines for legibility only):

```
[edit protocols ospf area area]
```



```
user@host# insert virtual-link neighbor-id 192.168.0.3 transit-area 1.1.1.1 \
        before virtual-link neighbor-id 192.168.0.5 transit-area 1.1.1.2
```

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <ospf>
          <area>
            <filter insert="before" neighbor-id="192.168.0.5" transit-area="1.1.1.2">
              <neighbor-id>192.168.0.3</neighbor-id>
              <transit-area>1.1.1.1</transit-area>
            </filter>
          </area>
        </ospf>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1180

Renaming a Configuration Object

To change the name of one or more of a configuration object's identifiers, a client application includes the tag elements described in "Creating, Modifying, or Deleting Configuration Elements" on page 117. In the object's opening tag, it includes the **rename="rename"** attribute and an attribute named after the identifier keyword. The value of the attribute is the new identifier value. The application includes the identifier tag element to specify the current name. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" name="new-name">
      <name>current-name</name>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

If the object has multiple identifiers, for each one the application includes both an attribute in the opening tag and an identifier tag element. If one or more of the identifiers is not changing, the attribute value for it is set to its current name. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" changing-identifier="new-name" \
          unchanging-identifier="current-name">
      <changing-identifier>current-name</changing-identifier>
      <unchanging-identifier>current-name</unchanging-identifier>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

The renaming operation is not available when formatted ASCII text is used to represent the configuration data.

The **rename** attribute can be combined with the **inactive** or **active** attribute to deactivate or reactivate the configuration element as it is renamed. For more information, see "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135.

The following example shows how to change the name of a firewall filter from **access-control** to **new-access-control**. This operation is equivalent to the following configuration mode command:

```
[edit firewall family inet]
user@host# rename filter access-control to filter new-access-control
```

Client Application **Junos XML Protocol Server**

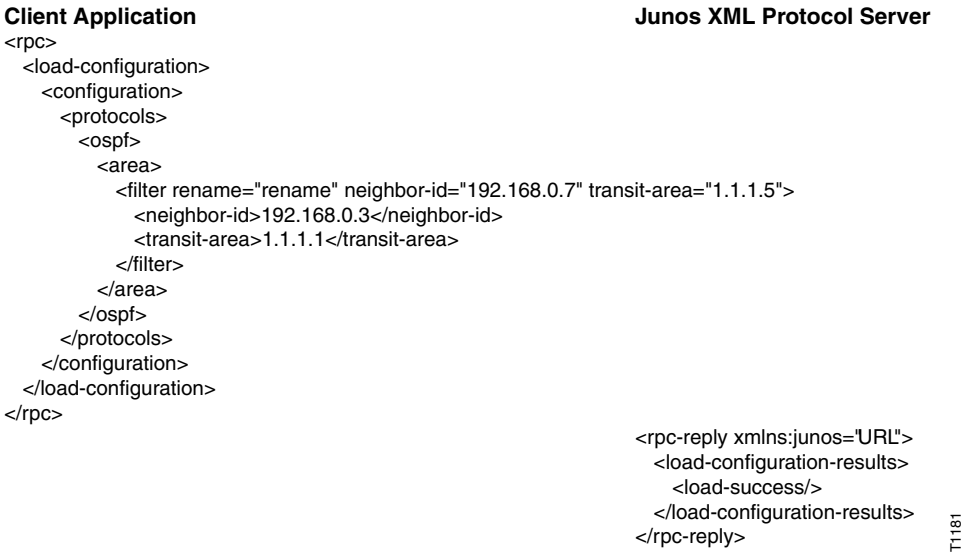
```
<rpc>
  <load-configuration>
    <configuration>
      <firewall>
        <family>
          <inet>
            <filter rename="rename" name="new-access-control">
              <name>access-control</name>
            </filter>
          </inet>
        </family>
      </firewall>
    </configuration>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1143

The following example shows how to change the identifiers for an OSPF virtual link (defined at the **[edit protocols ospf area *area*]** hierarchy level) from **neighbor-id 192.168.0.3** and **transit-area 1.1.1.1** to **neighbor-id 192.168.0.7** and **transit-area 1.1.1.5**. This operation is equivalent to the following configuration mode command (which appears on two lines for legibility only):

```
[edit protocols ospf area area]
user@host# rename filter virtual-link neighbor-id 192.168.0.3 transit-area \
1.1.1.1 to virtual-link neighbor-id 192.168.0.7 transit-area 1.1.1.5
```

Changing a Configuration Element's Activation State

When a configuration element (hierarchy level or configuration object) is deactivated, it remains in the candidate configuration or private copy, but when the configuration is later committed, the element does not affect the functioning of the routing platform. A client application can deactivate an element immediately as it creates it, or can deactivate an existing element. It can also reactivate an existing deactivated element so that when the configuration is committed, the element again has an effect on routing platform functioning.

See the following sections:

- Deactivating a Newly Created Element on page 133
- Deactivating or Reactivating an Existing Element on page 134

Deactivating a Newly Created Element

To define an element and immediately deactivate it, a client application includes the basic tag elements or configuration statements for its parent levels, as described in "Creating, Modifying, or Deleting Configuration Elements" on page 117.

If using Junos XML tag elements to define the new element, the application includes the **inactive="inactive"** attribute in the opening tag for the element. It includes tag elements for all children being defined for the element. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <element inactive="inactive">
      <name>identifier</name> <!-- if the element has an identifier -->
      <!-- tag elements for each child of the element -->
    </element>
  <!-- closing tag for each parent of the element -->
</configuration>
```

If using formatted ASCII text to define the new element, the application precedes the element with the **inactive:** operator. It includes all child statements that it is defining for all children of the element:

```
<configuration-text>
/* statements for parent levels */

  /* For an object with an identifier */
    inactive:
    object identifier {
      /* Child configuration statements */
    }

  /* For a hierarchy level or object without an identifier */
    inactive:
    element {
      /* Child configuration statements */
    }

/* closing braces for parent levels */
</configuration-text>
```

Deactivating or Reactivating an Existing Element

To deactivate an existing element, or reactivate a previously deactivated one, a client application includes the basic tag elements or configuration statements for its parent levels, as described in “Creating, Modifying, or Deleting Configuration Elements” on page 117.

If using Junos XML tag elements to represent a configuration object that has an identifier, the application includes the **inactive="inactive"** or **active="active"** attribute in the object’s opening container tag and also emits the identifier tag element and value. In the following, the identifier tag element is called **<name>**. To represent a hierarchy level or container object that has children but not an identifier, the application uses an empty tag:

```
<configuration>
    <!-- opening tag for each parent of the element -->
  !- - For an object with an identifier -->
    <object (inactive="inactive” | active="active")>
      <name>identifier</name>
    </object>

  <!-- For a hierarchy level or object without an identifier -->
    <level-or-container (inactive="inactive” | active="active")/>
  <!-- closing tag for each parent of the element -->
</configuration>
```

If using formatted ASCII text to represent the element, the application precedes the element with the **inactive:** or **active:** operator. The name of a hierarchy level or container object is followed by a semicolon (even though in the existing configuration it is followed by curly braces that enclose its child statements):

```
<configuration-text>
/* statements for parent levels */

  /* For an object with an identifier */
    (inactive | active):
```



```
        object identifier;

      /* For a hierarchy level or object without an identifier */
        (inactive | active):
        object-or-level;

    /* closing braces for parent levels */
    </configuration-text>
```

The following example shows how to deactivate the **isis** hierarchy level at the **[edit protocols]** hierarchy level in the candidate configuration.

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <isis inactive="inactive"/>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1145

Changing a Configuration Element's Activation State Simultaneously with Other Changes

A client application can deactivate or reactivate an element at the same time it performs other operations on it (except deletion), by combining the appropriate attributes or operators with the **inactive** or **active** attribute or operator. For basic information about activating or deactivating an element, see "Changing a Configuration Element's Activation State" on page 133.

To define the element to deactivate or activate, a client application includes the basic tag elements or configuration statements for its parent levels, as described in "Creating, Modifying, or Deleting Configuration Elements" on page 117. When using Junos XML tag elements to represent the element, the application includes the **inactive="inactive"** or **active="active"** attribute along with the appropriate other attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag. When using formatted ASCII text, the application combines the **inactive** or **active** operator with the other operator.

For instructions, see the following sections:

- Replacing an Element and Setting Its Activation State on page 136
- Reordering an Element and Setting Its Activation State on page 137
- Renaming an Object and Setting Its Activation State on page 137
- Example: Replacing an Object and Deactivating It on page 138

Replacing an Element and Setting Its Activation State

To replace (completely reconfigure) an element and simultaneously deactivate or activate it, a client application includes the tag elements or statements that represent all of the element's characteristics (for complete information about the syntax for defining elements, see "Replacing Configuration Elements" on page 121). The client application uses the attributes and operators discussed in the following sections to indicate which element is being replaced and activated or deactivated:

- Using Junos XML Tag Elements for the Replacement Element on page 136
- Using Formatted ASCII Text for the Replacement Element on page 136

Using Junos XML Tag Elements for the Replacement Element

If using Junos XML tag elements to represent the element, a client application includes the **action="replace"** attribute in the **<load-configuration>** tag element:

```
<rpc>
  <!-- For a file -->
    <load-configuration action="replace" url="file"/>

  <!-- For a data stream -->
    <load-configuration action="replace">
      <!-- Junos XML tag elements -->
    </load-configuration>
</rpc>
```

In the opening tag for the replacement element, the application includes two attributes—the **replace="replace"** attribute and either the **inactive="inactive"** or **active="active"** attribute. It includes tag elements for all children being defined for the element. In the following, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <element replace="replace" (inactive="inactive" | active="active")>
      <name>identifier</name> <!-- if the element has an identifier -->
      <!-- tag elements for each child of the element -->
    </element>
  <!-- closing tag for each parent of the element -->
</configuration>
```

Using Formatted ASCII Text for the Replacement Element

If using formatted ASCII text to represent the element, a client application includes the **action="replace"** and **format="text"** attributes in the **<load-configuration/>** tag or opening **<load-configuration>** tag:

```
<rpc>
  <!-- For a file -->
    <load-configuration action="replace" format="text" url="file"/>

  <!-- For a data stream -->
    <load-configuration action="replace" format="text">
      <!-- formatted ASCII configuration statements -->
    </load-configuration>
</rpc>
```

The application places the **inactive:** or **active:** operator on the line above the new element and the **replace:** operator directly before the new element. It includes all child statements that it is defining for all children of the element:

```
<configuration-text>
/* statements for parent levels */

    /* For an object with an identifier */
    (inactive | active):
    replace: object identifier {
      /* Child configuration statements */
    }

  /* For a hierarchy level or object without an identifier */
    (inactive | active):
    replace: element {
      /* Child configuration statements */
    }

/* closing braces for parent levels */
</configuration-text>
```

Reordering an Element and Setting Its Activation State

To reorder an element in an ordered list and simultaneously deactivate or activate it, the application combines the **insert** attribute and identifier attribute for the reference element with the **inactive** or **active** attribute. In the following, the identifier tag element for the moving element is called **<name>**. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="(before | after)" reference-identifier="value" \
          (inactive="inactive" | active="active")>
      <name>identifier-for-moving-object</name>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```

The reordering operation is not available when formatted ASCII text is used to represent the configuration data. For complete information about reordering elements, see “Reordering Elements in Configuration Objects” on page 129.

Renaming an Object and Setting Its Activation State

To rename an object (change the value of one or more of its identifiers) and simultaneously deactivate or activate it, the application combines the **rename** attribute and identifier attribute for the new name with the **inactive** or **active** attribute.

If the object has one identifier (here called **<name>**), the syntax is as follows (the opening tag appears on two lines for legibility only):

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" name="new-name" \
          (inactive="inactive” | active="active")>
      <name>current-name</name>
```

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

If the object has multiple identifiers and only one is changing, the syntax is as follows (the opening tag appears on multiple lines for legibility only):

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename"changing-identifier="new-name" \
          unchanging-identifier="current-name" \
          (inactive="inactive" | active="active")>
      <changing-identifier>current-name</changing-identifier>
      <unchanging-identifier>current-name</unchanging-identifier>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

The renaming operation is not available when formatted ASCII text is used to represent the configuration data. For complete information about renaming elements, see “Renaming a Configuration Object” on page 131.

Example: Replacing an Object and Deactivating It

The following example shows how to replace the information at the **[edit protocols bgp]** hierarchy level in the candidate configuration for the group called **G3**, and also deactivate the group so that it is not used in the actual configuration when the candidate is committed:

Client Application

```
<rpc>
  <load-configuration action="replace">
    <configuration>
      <protocols>
        <bgp>
          <group replace="replace" inactive="inactive">
            <name>G3</name>
            <type>external</type>
            <peer-as>58</peer-as>
            <neighbor>
              <name>10.0.20.1</name>
            </neighbor>
          </group>
        </bgp>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1146

The following example shows how to use formatted ASCII text to make the same changes:

Client Application **Junos XML Protocol Server**

```
<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      protocols {
        bgp {
        replace:
          inactive: group G3 {
            type external;
            peer-as 58;
            neighbor 10.0.20.1;
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1147

CHAPTER 6

Committing a Configuration

This chapter explains how to commit a 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 Junos XML protocol server confirms that it is syntactically correct. If the syntax check fails, the server does not commit the candidate. To avoid the potential complications of such a failure, it often makes sense to confirm the candidate's correctness before actually committing it. The client application encloses the empty **<check/>** tag in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <check/>
  </commit-configuration>
</rpc>
```

The Junos XML protocol server encloses its response in **<rpc-reply>**, **<commit-results>**, and **<routing-engine>** tag elements. If the syntax check succeeds, the **<routing-engine>** tag element encloses the **<commit-check-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the check succeeded (**re0** on routing platforms that use a single Routing Engine, and either **re0** or **re1** on routing platforms that can have two Routing Engines):

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
```

```
        <commit-check-success/>
      </routing-engine>
    </commit-results>
  </rpc-reply>
```

If the syntax check fails, an **<xnm:error>** tag element encloses tag elements that describe the error.

The **<check/>** tag can be combined with the **<synchronize/>** tag, which is described in "Verifying the Configuration on Both Routing Engines" on page 152.

Committing the Candidate Configuration

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

```
<rpc>
  <commit-configuration/>
</rpc>
```

We recommend that the application lock the candidate configuration before changing it and emit the **<commit-configuration/>** tag while the configuration is still locked. Doing so avoids inadvertent commit of changes made by other users or applications. After committing the configuration, the application must unlock it for other users and applications to be able to make changes. For instructions, see "Exchanging Information with the Junos XML Protocol Server" on page 48 and "Changing Configuration Information" on page 111.

The Junos XML protocol server reports the results of the commit operation in **<rpc-reply>**, **<commit-results>**, and **<routing-engine>** tag elements. If the commit operation succeeds, the **<routing-engine>** tag element encloses the **<commit-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the commit operation succeeded (**re0** on devices that use a single Routing Engine, and either **re0** or **re1** on devices that can have two Routing Engines):

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

If the commit operation fails, an **<xnm:error>** tag element encloses tag elements that describe the error. The most common causes of failure are semantic or syntactic errors in the candidate configuration.

Committing a Private Copy of the Configuration

To commit a private copy of the configuration so that it becomes the active configuration on the routing, switching, or security platform, a client application encloses the empty **<commit-configuration/>** tag in an **<rpc>** tag element (just as for the candidate configuration):

```
<rpc>
  <commit-configuration/>
</rpc>
```

The Junos XML protocol server creates a copy of the current regular candidate configuration, merges in the changes made to the private copy, and commits the combined candidate to make it the active configuration on the device. The server reports the results of the commit operation in **<rpc-reply>** and **<commit-results>** tag elements.

If the private copy does not include any changes, the server emits the **<commit-results>** and **</commit-results>** tags with nothing between them:

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
  </commit-results>
</rpc-reply>
```

If the private copy includes changes and the commit operation succeeds, the server emits the **<load-success/>** tag when it merges the changes in the private copy into the candidate configuration. The subsequent **<routing-engine>** tag element encloses the **<commit-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the commit operation succeeded (**re0** on devices that use a single Routing Engine, and either **re0** or **re1** on devices that can have two Routing Engines):

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <load-success/>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

If the private copy includes changes that conflict with the regular candidate configuration, the commit fails. The **<load-error-count>** tag element reports the number of errors and an **<xnm:error>** tag element encloses tag elements that describe the error.

There are restrictions on committing a private copy. For example, the commit fails if the regular candidate configuration is locked by another user or application, or if it includes uncommitted changes made since the private copy was created. For more information, see the *Junos OS CLI User Guide*.

Most of the variants of the commit operation are available for a private copy. The variants are described in subsequent sections in this chapter:

- Scheduling the commit for a later time, as described in “Committing a Configuration at a Specified Time” on page 144.

- Synchronizing the configuration on both Routing Engines, as described in "Committing and Synchronizing a Configuration on Redundant Control Planes" on page 148.
- Logging a commit-time message, as described in "Logging a Message About a Commit Operation" on page 154.

NOTE: The confirmed-commit operation is not available for a private copy. For information about using that operation for the regular candidate configuration, see "Committing the Candidate Configuration Only After Confirmation" on page 145.

Committing a Configuration at a Specified Time

To commit a configuration at a specified time in the future, a client application encloses the **<at-time>** tag element in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>
```

To indicate when to perform the commit operation, the application includes one of three types of values in the **<at-time>** tag element:

- The string **reboot**, to commit the configuration the next time the device reboots.
- A time value of the form ***hh:mm***[**:*ss***] (hours, minutes, and optionally seconds), to commit the configuration at the specified time, which must be after the time at which the application emits the **<commit-configuration>** tag element, but before 11:59:59 PM on the current day. For example, if the **<at-time>** tag element encloses the value **02:00** (2:00 AM) and the application emits the **<commit-configuration>** tag element at 2:10 AM, the commit will never take place, because the scheduled time has already passed for that day.

  Use 24-hour time; for example, **04:30:00** means 4:30:00 AM and **20:00** means 8:00 PM. The time is interpreted relative to the clock and time zone settings on the device..
- A date and time value of the form ***yyyy-mm-dd hh:mm***[**:*ss***] (year, month, date, hours, minutes, and optionally seconds), to commit the configuration at the specified day and time, which must be after the **<commit-configuration>** tag element is emitted. Use 24-hour time. For example, **2006-08-21 15:30:00** means 3:30 PM on August 21, 2006. The time is interpreted relative to the clock and time zone settings on the device.

NOTE: The specified time must be more than 1 minute later than the current time on the device.

The Junos XML protocol server immediately checks the configuration for syntactic correctness and returns **<rpc-reply>**, **<commit-results>**, and **<routing-engine>** tag elements. If the syntax check succeeds, the **<routing-engine>** tag element encloses the **<commit-check-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the check succeeded (**re0** on devices that use a single Routing Engine, and either **re0** or **re1** on devices that can have two Routing Engines). It also encloses an **<output>** tag element that reports the time at which the commit will occur:

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-check-success/>
      <output>commit at will be executed at timestamp</output>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

The configuration is scheduled for commit at the specified time. The Junos XML protocol server does not emit additional tag elements when it performs the actual commit operation.

If the configuration is not syntactically correct, an **<xnm:error>** tag element encloses tag elements that describe the error. The commit operation is not scheduled.

The **<at-time>** tag element can be combined with the **<synchronize/>** tag, the **<log/>** tag element, or both. For more information, see “Committing and Synchronizing a Configuration on Redundant Control Planes” on page 148 and “Logging a Message About a Commit Operation” on page 154.

The following example shows how to schedule a commit operation for 10:00 PM on the current day.

Client Application

```
<rpc>
  <commit-configuration>
    <at-time>22:00</at-time>
  </commit-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL">
  <commit-results>
    <routing-engine>
      <name>re1</name>
      <commit-check-success/>
      <output>commit at will be executed at  date 22:00:00  timezone</output>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

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 encloses the empty **<confirmed/>** tag in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
```

```
<commit-configuration>
  <confirmed/>
</commit-configuration>
</rpc>
```

If the commit is not confirmed within a certain amount of time (600 seconds [10 minutes] by default), the Junos XML protocol 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-configuration>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
  </commit-configuration>
</rpc>
```

NOTE: You cannot perform this commit operation on a private copy of the configuration.

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

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

If the Junos XML protocol server cannot commit the candidate, the **<rpc-reply>** tag element instead encloses an **<xnm: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.

In response to a confirmed commit operation, the Junos XML protocol server returns **<rpc-reply>**, **<commit-results>**, and **<routing-engine>** tag elements. If the commit operation succeeds, the **<routing-engine>** tag element encloses the **<commit-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the commit operation succeeded (**re0** on devices that use a single Routing Engine, and either **re0** or **re1** on devices that can have two Routing Engines):

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

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

To delay the rollback to a time later than the current rollback deadline, the application emits the **<confirmed/>** tag in a **<commit-configuration>** tag element again before the deadline passes. It can include 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 one of the following tag sequences before the rollback deadline passes:

- The empty **<commit-configuration/>** tag enclosed in an **<rpc>** tag element. The rollback is canceled and the candidate configuration is committed immediately, as described in "Committing the Candidate Configuration" on page 142. If the candidate configuration is still the same as the temporarily committed configuration, this effectively recommits the temporarily committed configuration:

  ```
  <rpc>
    <commit-configuration/>
  </rpc>
  ```

- The **<synchronize/>** tag enclosed in **<commit-configuration>** and **<rpc>** tag elements:

  ```
  <rpc>
    <commit-configuration>
      <synchronize/>
    </commit-configuration>
  </rpc>
  ```

  The rollback is canceled and the candidate configuration is checked and committed immediately on both Routing Engines, as described in "Committing and Synchronizing a Configuration on Redundant Control Planes" on page 148. If a confirmed commit operation has been performed on both Routing Engines, then emitting the **<synchronize/>** tag cancels the rollback on both.

- The **<at-time>** tag element enclosed in **<commit-configuration>** and **<rpc>** tag elements:

  ```
  <rpc>
    <commit-configuration>
      <at-time>time</at-time>
    </commit-configuration>
  </rpc>
  ```

  The rollback is canceled and the configuration is checked immediately for syntactic correctness, then committed at the scheduled time, as described in "Committing a Configuration at a Specified Time" on page 144.

The **<confirmed/>** and **<confirm-timeout>** tag elements can be combined with the **<synchronize/>** tag, the **<log/>** tag element, or both. For more information, see "Committing and Synchronizing a Configuration on Redundant Control Planes" on page 148 and "Logging a Message About a Commit Operation" on page 154.

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 Junos XML protocol 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 Junos XML Protocol Session" on page 57.

The following example shows how to commit the candidate configuration on Routing Engine 1 with a rollback deadline of 20 minutes.

Client Application

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

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re1</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1152

Committing and Synchronizing a Configuration on Redundant Control Planes

A Routing Engine resides within a control plane. For single-chassis configurations, there is one control plane. In redundant systems, there are two control planes, the master plane and the backup plane. In multichassis configurations, the control plane includes all Routing Engines with the same Routing Engine designation. For example, all master Routing Engines reside within the *master* control plane, and all backup Routing Engines reside within the *backup* control plane.

Committing a configuration applies a new configuration to the device Engine. In a multichassis configuration, once a change to the configuration has been committed to the system, this change is propagated throughout the control plane using the distribution function.

In a redundant architecture, you can issue the **synchronize** command to commit the new configuration to both the master and the slave control planes. When issued, this command will save the current configuration to both device Engines and commit the new configuration to both control planes. On a multichassis system, once the configuration has been committed on both planes, the distribution function will distribute the new configuration across both planes. For more information on Routing Engine redundancy, see the *Junos OS High Availability Configuration Guide*.

NOTE: In a multichassis architecture with redundant control planes, there is a difference between synchronizing the two planes and distributing the configuration throughout each plane. Synchronization only occurs between the Routing Engines within the same chassis. Once this synchronization is complete, the new configuration is distributed to all other Routing Engines within each plane as a separate distribution function.

Because synchronization happens across two separate control planes, synchronizing configurations is only valid on redundant Routing Engine architectures. Further, **re0** and **re1** configuration groups must be defined on each routing, swtiching, or security platform. For more information about configuration groups, see the *Junos OS CLI User Guide*.

NOTE: If you issue the **synchronize** command on a nonredundant Routing Engine system, the Junos XML protocol server will commit the configuration on the one control plane.

For more information about synchronizing configurations, see the following sections:

- Synchronizing the Configuration on Both Routing Engines on page 149
- Forcing a Synchronized Commit Operation on page 151
- Synchronizing Configurations Simultaneously with Other Operations on page 152

Synchronizing the Configuration on Both Routing Engines

To synchronize a configuration on a redundant Routing Engine system, a client application needs to enclose the empty **<synchronize/>** tag in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

The Junos XML protocol server verifies the configuration's syntactic correctness on the Routing Engine where the **<synchronize/>** tag is emitted (referred to as the local Routing Engine), copies the configuration to the remote Routing Engine and verifies its syntactic correctness there, and then commits the configuration on both Routing Engines.

The Junos XML protocol server encloses its response in **<rpc-reply>** and **<commit-results>** tag elements. It emits a separate **<routing-engine>** tag element for each operation on each Routing Engine:

- If the syntax check succeeds on a Routing Engine, the **<routing-engine>** tag element encloses the **<commit-check-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the check succeeded (**re0** or **re1**):

```
<routing-engine>
  <name>(re0 | re1)</name>
```

```
    <commit-check-success/>
  </routing-engine>
```

If the configuration is incorrect, an **<xnm:error>** tag element encloses a description of the error.

- If the commit operation succeeds on a Routing Engine, the **<routing-engine>** tag element encloses the **<commit-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the commit operation succeeded:

```
<routing-engine>
  <name>(re0 | re1)</name>
  <commit-success/>
</routing-engine>
```

If the commit operation fails, an **<xnm:error>** tag element encloses a description of the error. The most common causes of failure are semantic or syntactic errors in the configuration.

Example: Synchronizing the Configuration on Both Routing Engines

The following example shows how to commit and synchronize the candidate configuration on both Routing Engines.

Client Application

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-check-success/>
    </routing-engine>
    <routing-engine>
      <name>re1</name>
      <commit-check-success/>
    </routing-engine>
    <routing-engine>
      <name>re1</name>
      <commit-success/>
    </routing-engine>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1153

Forcing a Synchronized Commit Operation

The synchronize operation fails if the second Routing Engine's configuration is locked. If a synchronization failure occurs, it is best to determine the cause of the failure, take corrective action, and then synchronize the two Routing Engines again. However, when necessary, you can use the **<force-synchronize/>** command to override a locked configuration and force the synchronization.

NOTE: When you use a **force-synchronize** command, any uncommitted changes to the configuration will be lost.

To force a synchronization, enclose the empty **<synchronize/>** and **<force-synchronize/>** tags in the **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <force-synchronize/>
  </commit-configuration>
</rpc>
```

NOTE: In a multichassis environment, synchronization occurs between Routing Engines on the same chassis. Once the synchronization occurs, the configuration changes are propagated across each control plane using the distribution function. If one or more Routing Engines are locked during the distribution of the configuration, the distribution and thus the synchronization will fail. You will need to clear the error in the remote chassis and run the **synchronize** command again.

Example: Forcing a Synchronization

The following example shows how to force a synchronization across both Routing Engine planes:

Client Application	Junos XML Protocol Server
<rpc> <commit-configuration> <synchronize/> <force-synchronize/> <commit-configuration> </rpc>	
	<rpc-reply xmlns:junos= ”http://xml.juniper.net/junos/9.010/junos”> <commit-results> <routing-engine junos:style=”show-name”> <name>re0</name> <commit-check-success/> </routing-engine> <routing-engine junos:style=”show-name”> <name>re1</name> <commit-success/> </routing-engine> <routing-engine junos:style=”show-name”> <name>re0</name> <commit-success/> </routing-engine> </commit-resuls> </rpc-reply>

Synchronizing Configurations Simultaneously with Other Operations

The **<synchronize/>** tag can be combined with the other tag elements that can occur within the **<commit-configuration>** tag element. The Junos XML protocol server checks, copies, and commits the configuration, and emits the same response tag elements as when the **<synchronize/>** tag is used by itself. The possible combinations are described in the following sections.

Verifying the Configuration on Both Routing Engines

To check the syntactic correctness of a local configuration on both Routing Engines without committing it, the application encloses the **<synchronize/>** and **<check/>** tag elements in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <check/>
  </commit-configuration>
</rpc>
```

The **<force-synchronize/>** tag cannot be combined with the **<check/>** tag elements.

For more information about verifying configurations, see "Verifying a Configuration Before Committing It" on page 141.

Scheduling Synchronization for a Specified Time

To commit a configuration on both Routing Engines at a specified time in the future, the application encloses the **<synchronize/>** and **<at-time>** tag elements in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>

<rpc>
  <commit-configuration>
    <force-synchronize/>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>
```

As when the **<at-time>** tag element is emitted by itself, the Junos XML protocol server verifies syntactic correctness immediately and does not emit additional tag elements when it actually performs the commit operation on each Routing Engine. For information about how to specify the time in the **<at-time>** tag element, see "Committing the Candidate Configuration Only After Confirmation" on page 145.

Synchronizing Configurations but Requiring Confirmation

To commit the candidate configuration on both Routing Engines but require confirmation for the commit to become permanent, the application encloses the **<synchronize/>**, **<confirmed/>**, and (optionally) **<confirm-timeout>** tag elements in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <confirmed/>
      [<confirm-timeout>minutes</confirm-timeout>]
  </commit-configuration>
</rpc>
```

The same rollback deadline applies to both Routing Engines and can be extended on both at once by again emitting the **<synchronize/>**, **<confirmed/>**, and (optionally) **<confirm-timeout>** tag elements on the Routing Engine where the tag elements were emitted the first time.

The **<force-synchronize/>** tag cannot be combined with the **<confirmed/>** and **<confirm-timeout>** tag elements.

For more information about confirmed commit operations, see "Committing the Candidate Configuration Only After Confirmation" on page 145.

Logging a Message About Synchronized Configurations

To synchronize configurations and record a log message when the commit succeeds on each Routing Engine, the application encloses the **<synchronize/>** and **<log/>** tag elements in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <log>message</log>
  </commit-configuration>
</rpc>
<rpc>
  <commit-configuration>
    <force-synchronize/>
    <log>message</log>
  </commit-configuration>
</rpc>
```

The commit operation proceeds as previously described in the **<synchronize/>** or **<force-synchronize/>** tag descriptions. The message for each Routing Engine is recorded in the separate **/var/log/commits** file maintained by that Routing Engine. For more information about logging, see "Logging a Message About a Commit Operation" on page 154.

Logging a Message About a Commit Operation

To record a message in the **/var/log/commits** file when a commit operation succeeds, a client application encloses the **<log>** tag element in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <log>message</log>
  </commit-configuration>
</rpc>
```

The **<log>** tag element can be combined with other tag elements within the **<commit-configuration>** tag element (the **<at-time>**, **<confirmed/>**, and **<confirm-timeout>**, or **<synchronize/>** tag elements) and does not change the effect of the operation. When the **<log>** tag element is emitted by itself, the associated commit operation begins immediately.

The following example shows how to log a message as the candidate configuration is committed.

Client Application

```
<rpc>
  <commit-configuration>
    <log>Enable xnm-ssl service</log>
  </commit-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos='URL'>
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1154

The **/var/log/commits** file includes an entry for each pending commit and up to 50 previous commits. To request the contents of the file, a client application encloses the **<get-commit-information/>** tag in **<rpc>** tag elements:

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

(The equivalent operational mode CLI command is **show system commit**.) The Junos XML protocol server encloses the information in **<commit-information>** and **<rpc-reply>** tag elements. For information about the child tag elements of the **<commit-information>** tag element, see its entry in the *Junos XML API Operational Reference*.

```
<rpc-reply xmlns:junos="URL">
  <commit-information>
    <!-- tag elements representing the commit log -->
  </commit-information>
</rpc-reply>
```

The following example shows how to request the commit log.

Client Application **Junos XML Protocol Server**

```
<rpc>
  <get-commit-information/>
</rpc>
                <rpc-reply xmlns:junos="URL">
                  <commit-information>
                    <commit-history>
                      <sequence-number>0</sequence-number>
                      <user>barbara</user>
                      <client>other</client>
                      <date-time junos:seconds="1058370173">2003-07-16 08:42:53 PDT</date-time>
                      <log>Enable xnm-ssl service</log>
                    </commit-history>
                    <commit-history>
                      <sequence-number>1</sequence-number>
                      <user>root</user>
                      <client>other</client>
                      <date-time junos:seconds="1058322166">2003-07-15 19:22:46 PDT</date-time>
                    </commit-history>
                    <commit-history>
                      <sequence-number>2</sequence-number>
                      <user>root</user>
                      <client>cli</client>
                      <date-time junos:seconds="1058219717">2003-07-14 14:55:17 PDT</date-time>
                    </commit-history>
                          .
                          .
                          .
                  </commit-information>
                </rpc-reply>
```

T1183

CHAPTER 7

Summary of Junos XML Protocol Tag Elements

This chapter lists the tag elements that client applications and the Junos XML protocol server use to control the Junos XML protocol session and to exchange configuration information. The entries are in alphabetical order. For information about the notational conventions used in this chapter, see Table 2 on page xxii.

<abort/>

Usage

```
<rpc>
  <!-- child tag elements -->
</rpc>
<abort/>
```

Description Direct the Junos XML protocol server to stop processing the request that is currently outstanding. The server responds by returning the **<abort-acknowledgment/>** tag, but might already have sent tagged data in response to the request. The client application must discard those tag elements.

Usage Guidelines See "Halting a Request" on page 54.

Related Documentation

- <abort-acknowledgement/> on page 157
- <rpc> on page 179

<abort-acknowledgement/>

Usage

```
<rpc-reply xmlns:junos="URL">
  <any-child-of-rpc-reply>
    <abort-acknowledgement/>
  </any-child-of-rpc-reply>
</rpc-reply>
```

Description Indicate that the Junos XML protocol server has received the **<abort/>** tag and has stopped processing the current request. If the client application receives any tag elements related to the request between sending the **<abort/>** tag and receiving this tag, it must discard them.

Usage Guidelines See "Halting a Request" on page 54.

Related Documentation
- <rpc-reply> on page 180
- <xnm:error> on page 181

<authentication-response>

Usage

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>authentication-outcome</status>
    <message>message</message>
    <login-name>remote-username</login-name>
  </authentication-response>
</rpc-reply>
```

Description Indicate whether an authentication attempt succeeded. The Junos XML protocol server returns the tag element in response to the **<request-login>** tag element emitted by a client application that uses the clear-text or Secure Sockets Layer (SSL) access protocol.

Contents

<login-name>—Specifies the username that the client application provided to an authentication utility such as RADIUS or TACACS+. This tag element appears only if the username that it contains differs from the username contained in the **<message>** tag element.

<message>—Names the account under which a connection to the Junos XML protocol server is established, if authentication succeeds. If authentication fails, explains the reason for the failure.

<status>—Indicates whether the authentication attempt succeeded. There are two possible values:

- **fail**—The attempt failed. The Junos XML protocol server also emits the **<challenge>** tag element to request the password again, up to a maximum of three attempts.
- **success**—The attempt succeeded. An authenticated connection to the Junos XML protocol server is established.

Usage Guidelines See "Interpreting the Authentication Response" on page 47.

Related Documentation
- <challenge> on page 158
- <request-login> on page 178
- <rpc-reply> on page 180

<challenge>

Usage

```
<rpc-reply xmlns:junos="URL">
  <challenge echo="no">Password:</challenge>
```

```
</rpc-reply>
```

Description Request the password associated with an account during authentication with a client application that uses the clear-text or SSL access protocol. The Junos XML protocol server emits this tag element when the initial **<request-login>** tag element emitted by the client application does not enclose a **<challenge-response>** tag element, and when the password enclosed in a **<challenge-response>** tag element is incorrect (in the latter case, the server also emits an **<authentication-response>** tag element enclosing child tag elements that indicate the password is incorrect).

The tag element encloses the string **Password:** which the client application can forward to the screen as a prompt for a user.

Attributes **echo**—Specifies whether the password string typed by the user appears on the screen. The value **no** specifies that it does not.

Usage Guidelines See "Submitting an Authentication Request" on page 46.

Related Documentation

- <authentication-response> on page 158
- <request-login> on page 178
- <rpc-reply> on page 180

<checksum-information>

Usage

```
<rpc-reply>
  <checksum-information>
    <file-checksum>
      <computation-method>MD5</computation-method>
      <input-file>
        <!-- name and path of file-->
      </input-file>
    </file-checksum>
  </checksum-information>
</rpc-reply>
```

Description Enclose tag elements that include the file to check, the checksum algorithm used, and the checksum output.

Contents **<file-checksum>**—Wrapper that holds the resulting **<input-file>**, **<computation-method>**, and **<checksum>** attributes for a particular checksum computation.

<input-file>—Name and path of the file that the checksum algorithm was run against.

<computation-method>—Checksum algorithm used. Currently, all checksum computations use the MD5 algorithm; thus, the only possible value is MD5.

<checksum>—Resulting value from the checksum computation.

Usage Guidelines See the *Junos XML API Operational Reference*.

Related Documentation
- <get-checksum-information> on page 167

<close-configuration/>

Usage

```
<rpc>
  <close-configuration/>
</rpc>
```

Description Discard a candidate configuration and any changes to it.

This tag element is normally used only to discard a private copy of the candidate configuration without committing it. The application must have previously emitted the **<open-configuration>** tag element. Closing the Junos XML protocol session (by emitting the **<request-end-session/>** tag, for example) has the same effect as emitting this tag element.

Usage Guidelines See "Creating a Private Copy of the Configuration" on page 58.

Related Documentation
- <open-configuration> on page 176
- <request-end-session/> on page 177
- <rpc> on page 179

<commit-configuration>

Usage

```
<rpc>
  <commit-configuration/>

  <commit-configuration>
    <check/>
  </commit-configuration>

  <commit-configuration>
    <log>log-message</log>
  </commit-configuration>

  <commit-configuration>
    <at-time>time-specification</at-time>
    <log>log-message</log>
  </commit-configuration>

  <commit-configuration>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
    <log>log-message</log>
  </commit-configuration>
```



```
<commit-configuration>
  <synchronize/>
  <log>log-message</log>
</commit-configuration>

<commit-configuration>
  <synchronize/>
  <at-time>time-specification</at-time>
  <log>log-message</log>
</commit-configuration>

<commit-configuration>
  <synchronize/>
  <check/>
  <log>log-message</log>
</commit-configuration>

<commit-configuration>
  <synchronize/>
  <confirmed/>
  <confirm-timeout>rollback-delay</confirm-timeout>
  <log>log-message</log>
</commit-configuration>

<commit-configuration>
  <synchronize/>
  <force-synchronize/>
</commit-configuration>
</rpc>
```

Description Request that the Junos XML protocol server perform one of the variants of the commit operation on either the regular candidate configuration or a private copy of the candidate configuration (if the application emitted the **<open-configuration><private/></open-configuration>** tag sequence before making changes).

Some restrictions apply to the commit operation for a private copy. For example, the commit operation fails if the regular candidate configuration is locked by another user or application, or if it includes uncommitted changes made since the private copy was created. For more information, see the *Junos OS CLI User Guide*.

Enclose the appropriate tag in the **<commit-configuration>** tag element to specify the type of commit operation:

- To commit the configuration immediately, making it the active configuration on the device, emit the empty **<commit-configuration/>** tag.
- To verify the syntactic correctness of the configuration without actually committing it, enclose the **<check/>** tag in the **<commit-configuration>** tag element.
- To record a message in the **/var/log/commits** file when the associated commit operation succeeds, define the log message string in the **<log>** tag element and enclose the tag element in the **<commit-configuration>** tag element. The **<log>** tag element can be

combined with any other tag element. When the **<log>** tag element is emitted alone, the associated commit operation begins immediately.

- To commit the candidate configuration but roll back to the previous configuration after a short time, enclose the **<confirmed/>** tag in the **<commit-configuration>** tag element.

  By default, the rollback occurs 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-configuration>** tag element) 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-configuration/>** tag before the rollback deadline passes. The Junos XML protocol server commits the candidate configuration and cancels the rollback. If the candidate configuration is still the same as the current committed configuration, the effect is the same as recommitting the current committed configuration.

  NOTE: The confirmed commit operation is not available for a private copy of the configuration.

- On a device with two Routing Engines, commit the candidate configuration stored on the local Routing Engine on both Routing Engines. Combine tag elements as indicated in the following:

  - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it immediately on both Routing Engines, enclose the **<synchronize/>** tag in the **<commit-configuration>** tag element.

  - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines at a defined future time, enclose the **<synchronize/>** or **<force-synchronize/>** tag and **<at-time>** tag element in the **<commit-configuration>** tag element. Set the value in the **<at-time>** tag element as previously described for use of the **<at-time>** tag element alone.

  - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine and verify the candidate's syntactic correctness on each Routing Engine, enclose the **<synchronize/>** or **<force-synchronize/>** and **<check/>** tag elements in the **<commit-configuration>** tag element.

  - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines but require confirmation, enclose the **<synchronize/>** tag and **<confirmed/>** tag elements, and optionally the **<confirm-timeout>** tag element, in

the **<commit-configuration>** tag element. Set the value in the **<confirm-timeout>** tag element as previously described for use of the **<confirmed/>** tag and **<confirm-timeout>** tag element alone.

- To force the same synchronized commit operation as invoked by the **<synchronize/>** tag to succeed, even if there are open configuration sessions or uncommitted configuration changes on the remote machine, enclose the **<force-synchronize/>** tag in the **<commit-configuration>** tag element.

- To schedule the configuration for commit at a future time, enclose the **<at-time>** tag element in the **<commit-configuration>** tag element. There are three valid types of time specifiers:
  - The string **reboot**, to commit the configuration the next time the device reboots.
  - A time value of the form ***hh*:*mm*[:*ss*]** (hours, minutes, and, optionally, seconds), to commit the configuration at the specified time, which must be in the future but before 11:59:59 PM on the day the **<commit-configuration>** tag element is emitted. Use 24-hour time for the ***hh*** value; for example, **04:30:00** means 4:30:00 AM and **20:00** means 8:00 PM. The time is interpreted with respect to the clock and time zone settings on the device.
  - A date and time value of the form ***yyyy-mm-dd hh*:*mm*[:*ss*]** (year, month, date, hours, minutes, and, optionally, seconds), to commit the configuration at the specified date and time, which must be after the **<commit-configuration>** tag element is emitted. Use 24-hour time for the ***hh*** value. For example, **2005-08-21 15:30:00** means 3:30 PM on August 21, 2005. The time is interpreted with respect to the clock and time zone settings on the device.

    NOTE: The time you specify must be more than 1 minute later than the current time on the device.

    The configuration is checked immediately for syntactic correctness. If the check succeeds, the configuration is scheduled for commit at the specified time. If the check fails, the commit operation is not scheduled.

Contents **<at-time>**—Schedules the commit operation for a specified future time.

<check>—Requests verification that the configuration is syntactically correct, but does not actually commit it.

<confirmed>—Requests a commit of the candidate configuration and a rollback to the previous configuration after a short time, 10 minutes by default. Use the **<confirm-timeout>** tag element to specify a different amount of time.

<confirm-timeout>—Specifies the number of minutes for which the configuration remains active when the **<confirmed/>** tag is enclosed in the **<commit-configuration>** tag element.

<log>—Records a message in the file **/var/log/commits** when the commit operation succeeds.

<synchronize>—On dual control plane systems, requests that the candidate configuration on one control plane be copied to the other control plane, checked for correct syntax, and committed on both Routing Engines.

<force-synchronize>—On dual control plane systems, forces the candidate configuration on one control plane to be copied to the other control plane.

Usage Guidelines See "Committing a Configuration" on page 141.

Related Documentation
- <commit-results> on page 164
- <open-configuration> on page 176
- <rpc> on page 179

<commit-results>

Usage

```
<rpc-reply xmlns:junos="URL">
  <!-- for the candidate configuration -->
  <commit-results>
    <routing-engine>...</routing-engine>
  </commit-results>

  <!-- for a private copy -->
  <commit-results>
    <load-success/>
    <routing-engine>...</routing-engine>
  </commit-results>

  <!-- for a private copy that does not include changes -->
  <commit-results>
  </commit-results>
</rpc-reply>
```

Description Enclose tag elements that contain information about a commit operation performed by the Junos XML protocol server on a particular Routing Engine.

Contents **<load-success/>**—Indicates that the Junos XML protocol server successfully merged changes from the private copy into a copy of the candidate configuration, before committing the combined candidate on the specified Routing Engine.

The **<routing-engine>** tag element is described separately.

Usage Guidelines See "Committing a Configuration" on page 141.

Related Documentation
- <commit-configuration> on page 160
- <routing-engine> on page 178

- <rpc-reply> on page 180

<database-status>

Usage

```
<junoscript>
  <any-child-of-junoscript>
    <xnm:error>
      <database-status-information>
        <database-status>
          <user>username</user>
          <terminal>terminal</terminal>
          <pid>pid</pid>
          <start-time>start-time</start-time>
          <idle-time>idle-time</idle-time>
          <commit-at>time</commit-at>
          <exclusive/>
          <edit-path>edit-path</edit-path>
        </database-status>
      </database-status-information>
    </xnm:error>
  </any-child-of-junoscript>
</junoscript>
```

Description Describe a user or Junos XML protocol client application that is logged in to the configuration database. For simplicity, the Contents section uses the term user to refer to both human users and client applications, except where the information differs for the two.

Contents **<commit-at/>**—Indicates that the user has scheduled a commit operation for a later time.

<edit-path>—Specifies the user's current location in the configuration hierarchy, in the form of the CLI configuration mode banner.

<exclusive/>—Indicates that the user or application has an exclusive lock on the configuration database. A user enters exclusive configuration mode by issuing the **configure exclusive** command in CLI operational mode. A client application obtains the lock by emitting the **<lock-configuration/>** tag element.

<idle-time>—Specifies how much time has passed since the user last performed an operation in the database.

<pid>—Specifies the process ID of the Junos management process (mgd) that is handling the user's login session.

<start-time>—Specifies the time when the user logged in to the configuration database, in the format ***YYYY-MM-DD hh:mm:ss TZ*** (year, month, date, hour in 24-hour format, minute, second, time zone).

<terminal>—Identifies the UNIX terminal assigned to the user's connection.

<user>—Specifies the Junos OS login ID of the user whose login to the configuration database caused the error.

Usage Guidelines See "Handling an Error or Warning" on page 53.

Related Documentation
- <database-status-information> on page 166
- <junoscript> on page 170
- <xnm:error> on page 181

<database-status-information>

Usage

```
<junoscript>
  <any-child-of-junoscript>
    <xnm:error>
      <database-status-information>
        <database-status>...</database-status>
      </database-status-information>
    <xnm:error>
  </any-child-of-junoscript>
<junoscript>
```

Description Describe one or more users who have an open editing session in the configuration database.

The **<database-status>** tag element is explained separately.

Usage Guidelines See "Handling an Error or Warning" on page 53.

Related Documentation
- <database-status> on page 165
- <junoscript> on page 170
- <xnm:error> on page 181

<end-session/>

Usage

```
<rpc-reply xmlns:junos="URL">
  <end-session/>
</rpc-reply>
```

Description Indicate that the Junos XML protocol server is about to end the current session for a reason other than an error. Most often, the reason is that the client application has sent the **<request-end-session/>** tag.

Usage Guidelines See "Ending a Junos XML Protocol Session and Closing the Connection" on page 58.

Related Documentation
- <request-end-session/> on page 177
- <rpc-reply> on page 180

<get-checksum-information>

Usage

```
<rpc>
  <get-checksum-information>
    <path>
    <!-- name and path of file -->
    </path>
  </get--checksum-information>
</rpc>
```

Description Enclose all tag elements in a request generated by a client application.

Contents **<path>**—The name and path of the file to check.

Usage Guidelines See the *Junos XML API Operational Reference*.

Related Documentation

- <checksum-information> on page 159

<get-configuration>

Usage

```
<rpc>
  <get-configuration
            [changed="changed"]
            [commit-scripts="view"]
            [compare="rollback" [rollback="[0-49]"]]
            [database="(candidate | committed)"]
            [format="(text | xml)"]
            [inherit="(defaults | inherit)"
                  [groups="groups"] [interface-ranges="interface-ranges"]]/>

    <!-- tag elements for the configuration element to display -->
  </get-configuration>
</rpc>
```

Description Request configuration data from the Junos XML protocol server. The attributes specify the source and formatting of the data to display. Either the entire configuration hierarchy or a section can be displayed:

- To display the entire configuration hierarchy, emit the empty **<get-configuration/>** tag.
- To display a configuration element (hierarchy level or configuration object), emit tag elements within the **<get-configuration>** tag element to represent all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the level or object to display. To represent a hierarchy level or a configuration object that does not have an identifier, emit it as an empty tag. To represent an object that has one or more identifiers, emit its container tag element and identifier tag elements only, not any tag elements that represent other characteristics.

Attributes **changed**—Specifies that the **junos:changed="changed";** attribute should appear in the opening tag of each changed configuration element.

The attribute appears in the opening tag of every parent tag element in the path to the changed configuration element, including the top-level opening **<configuration>** tag. If the changed configuration element is represented by a single (empty) tag, the **junos:changed="changed"** attribute appears in the tag. If the changed element is represented by a container tag element, the **junos:changed="changed"** attribute appears in the opening container tag and also in each child tag element enclosed in the container tag element.

The **database** attribute can be combined with the **changed="changed"** attribute to request either the candidate or active configuration:

- When the candidate configuration is requested (the **database="changed"** attribute is included or the **database** attribute is omitted completely), elements added to the candidate configuration after the last commit operation are marked with the **junos:changed="changed"** attribute.
- When the active configuration is requested (the **database="candidate"** attribute is included), elements added to the active configuration by the most recent commit are marked with the **junos:changed="changed"** attribute.

NOTE: When a commit operation succeeds, the Junos XML protocol server removes the junos:changed="changed" attribute from all tag elements. However, if warnings are generated during the commit, the attribute is not removed. In this case, the junos:changed="changed" attribute appears in tag elements that changed before the commit operation as well as on those that changed after it.

An example of a commit-time warning is the message explaining that a configuration element will not actually apply until the device is rebooted. The warning appears in the tag string that the Junos XML protocol server returns to confirm the success of the commit, enclosed in an **<xnm:warning>** tag element.

To remove the **junos:changed="changed"** attribute from elements that changed before the commit, take the action necessary to eliminate the cause of the warning, and commit the configuration again.

commit-scripts—Requests that the Junos XML protocol server display commit-script-style XML data, which displays the configuration in the XML format that is input to a commit script. The only acceptable value for the **commit-scripts** attribute is **view**. The output is equivalent to the CLI output when using the **| display commit-scripts view** option.

compare—Requests that the Junos XML protocol server display the differences between the active or candidate configuration and a previously committed configuration. The only acceptable value for the **compare** attribute is **rollback**. The compare attribute is combined with the **rollback="*rollback-number*"** to specify which previously committed configuration

should be used in the comparison. If the **rollback** attribute is omitted, the comparison uses rollback number 0, which is the active configuration.

The **database** attribute can be combined with the **compare="rollback"** attribute to request either the candidate or active configuration. If the **database** attribute is omitted, the candidate configuration is used. When the **compare** attribute is used, the default format for the output is text. If the client application attempts to include the **format="xml"** attribute when the **compare="rollback"** attribute is present, the protocol server will return an **<xnm:error>** element indicating an error.

database—Specifies the version of the configuration from which to display data. There are two acceptable values:

- **candidate**—The candidate configuration
- **committed**—The active configuration (the one most recently committed)

format—Specifies the format in which the Junos XML protocol server returns the configuration data. There are two acceptable values:

- **text**—Configuration statements are formatted as ASCII text, using the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements. This is the format used in configuration files stored on a device running Junos OS and displayed by the CLI **show configuration** command.
- **xml**—Configuration statements are represented by the corresponding Junos XML tag elements. This is the default value if the **format** attribute is omitted.

groups—Specifies that the **junos:group="*group-name*"** attribute appears in the opening tag for each configuration element that is inherited from a configuration group. The ***group-name*** variable specifies the name of the configuration group.

The **groups** attribute must be combined with the **inherit** attribute, and the one acceptable value for it is **groups**.

inherit—Specifies how the Junos XML protocol server displays statements that are defined in configuration groups and interface ranges. If the **inherit** attribute is omitted, the output uses the **<groups>**, **<apply-groups>**, and **<apply-groups-except>** tag elements to represent user-defined configuration groups and uses the **<interface-range>** tag element to represent user-defined interface ranges; it does not include tag elements for statements defined in the **junos-defaults** group.

There are two acceptable values:

- **defaults**—The output does not include the **<groups>**, **<apply-groups>**, and **<apply-groups-except>** tag elements, but instead displays tag elements that are inherited from user-defined groups and from the **junos-defaults** group as children of the inheriting tag elements.
- **inherit**—The output does not include the **<groups>**, **<apply-groups>**, **<apply-groups-except>**, and **<interface-range>** tag elements, but instead displays tag elements that are inherited from user-defined groups and ranges as children of the inheriting tag elements. The output does not include tag elements for statements defined in the **junos-defaults** group.

interface-ranges—Specifies that the **junos:interface-ranges="*source-interface-range*"** attribute appears in the opening tag for each configuration element that is inherited from an interface-range. The ***source-interface-range*** variable specifies the name of the interface-range.

The **interface-ranges** attribute must be combined with the **inherit** attribute, and the one acceptable value for it is **interface-ranges**.

Usage Guidelines See “Requesting Configuration Information” on page 68.

Related Documentation

- junos:changed on page 189
- junos:group on page 192
- junos:interface-range on page 192
- <rpc> on page 179
- <xnm:warning> on page 183
- *Junos XML API Configuration Reference*

<junoscript>

Usage

```
<!-- emitted by a client application -->
<junoscript version="version" [hostname="hostname"] [junos:key="key"]
            [release="release"]>
  <!-- all tag elements generated by the application during the session -->
</junoscript>

<!-- emitted by the Junos XML protocol server -->
<junoscript xmlns="namespace-URL" xmlns:junos="namespace-URL"
            schemaLocation="namespace-URL" os="os" release="release"
            hostname="hostname" version="version">
  <!- - all tag elements generated by the Junos XML protocol server during the session -
->
</junoscript>
```


Description Enclose all tag elements in a Junos XML protocol session (act as the root tag element for the session). The client application and Junos XML protocol server each emit this tag element, enclosing all other tag elements that they emit during a session inside it.

Attributes **hostname**—Names the machine on which the tag element's originator is running.

junos:key—Requests that the Junos XML protocol server include the **junos:key="key"** attribute in the opening tag of each tag element that serves as an identifier for a configuration object.

os—Specifies the operating system of the machine named by the **hostname** attribute.

release—Identifies the Junos OS Release being run by the tag element's originator. Software modules always set this attribute, but client applications are not required to set it.

schemaLocation—Specifies the XML namespace for the XML Schema-language representation of the Junos configuration hierarchy.

version—(Required for a client application) Specifies the version of the Junos XML management protocol used for the enclosed set of tag elements.

xmlns—Names the XML namespace for the tag elements enclosed by the **<junoscript>** tag element that do not have a prefix on their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form **http://xml.juniper.net/xnm/*version-code*/xnm**, where ***version-code*** is a string such as **1.1**.

xmlns:junos—Names the XML namespace for the tag elements enclosed by the **<junoscript>** tag element that have the **junos:** prefix. The value is a URL of the form **http://xml.juniper.net/junos/*release-code*/junos**, where ***release-code*** is the standard string that represents a release of the Junos OS, such as **10.4R1** for the initial version of Junos OS Release 10.4.

Usage Guidelines See "Emitting the Opening <junoscript> Tag" on page 42, "Parsing the Junos XML Protocol Server's Opening <junoscript> Tag" on page 44, and "Requesting an Indicator for Identifiers" on page 74.

Related Documentation

- <rpc> on page 179
- <rpc-reply> on page 180
- junos:key on page 193

<kill-session>

Usage

```
<rpc>
  <kill-session>
    <session-id>PID</session-id>
  </kill-session>
</rpc>
```

Description Request that the Junos XML protocol server terminate another CLI or Junos XML protocol 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 **<xnm:error>** tag element that the Junos XML protocol server generates when it cannot lock a configuration as requested.

Usage Guidelines "Terminating Another Junos XML Protocol Session" on page 57

Related Documentation
- <lock-configuration/> on page 176
- <xnm:error> on page 181

<load-configuration>

Usage

```
<rpc>
  <load-configuration rescue="rescue"/>

  <load-configuration rollback="index"/>

  <load-configuration url="url " [action="(merge | override | replace | update)"] \
          [format="(text | xml)"]/>

  <load-configuration format="text" [action="(merge | override | replace | \
          update)"]>
    <configuration-text>
      <!-- formatted ASCII configuration statements to load -->
    </configuration-text>
  </load-configuration>

  <load-configuration [action="(merge | override | replace | update)"] \
          [format="xml"]>
    <configuration>
      <!-- tag elements for configuration elements to load -->
    </configuration>
  </load-configuration>
</rpc>
```


Description Request that the Junos XML protocol server load configuration data into the candidate configuration. Provide the data to load in one of four ways:

- Set the empty **<load-configuration/>** tag's **rescue** attribute to the value **rescue**. The rescue configuration completely replaces the candidate configuration.
- Set the empty **<load-configuration/>** tag's **rollback** attribute to the numerical index of a previous configuration. The routing platform stores a copy of the most recently committed configuration and up to 49 previous configurations. The specified previous configuration completely replaces the candidate configuration.
- Set the empty **<load-configuration/>** tag's **url** attribute to the pathname of a file that contains the configuration data to load. The data can be either formatted ASCII text (in which case the **format** attribute must be set to the value **text**) or Junos XML tag elements (in which case the **format** attribute is either omitted or set to the value **xml**).

  In the following example, the **url** attribute identifies **/tmp/add.conf** as the file to load.

  ```
  <load-configuration url="/tmp/add.conf"/>
  ```

- Enclose the configuration data within an opening **<load-configuration>** and closing **</load-configuration>** tag. If providing the configuration data as formatted ASCII text, enclose it in a **<configuration-text>** tag element and set the **format** attribute to the value **text**. If providing configuration data as Junos XML tag elements, enclose it in a **<configuration>** tag element and optionally set the **format** attribute to the value **xml**.

Attributes **action**—Specifies how to load the configuration data, particularly when the candidate configuration and loaded configuration contain conflicting statements. The following are acceptable values:

- **merge**—Combines the data in the loaded configuration with the candidate configuration. If statements in the loaded configuration conflict with statements in the candidate configuration, the loaded statements replace the candidate ones. This is the default behavior if the **action** attribute is omitted.
- **override**—Discards the entire candidate configuration and replaces it with the loaded configuration. When the configuration is later committed, all system processes parse the new configuration.
- **replace**—Substitutes each hierarchy level or configuration object defined in the loaded configuration for the corresponding level or object in the candidate configuration.

  If providing the configuration data as formatted ASCII text (either in the file named by the **url** attribute or enclosed in a **<configuration-text>** tag element), also place the **replace:** statement on the line directly preceding the statements that represent the hierarchy level or object to replace. For more information, see the discussion of loading a file of configuration data in the *Junos OS CLI User Guide*.

  If providing the configuration data as Junos XML tag elements, also set the **replace** attribute to the value **replace** on the opening tag of the container tag element that represents the hierarchy level or object to replace.
- **update**—Compares the loaded configuration and candidate configuration. For each hierarchy level or configuration object that is different in the two configurations, the version in the loaded configuration replaces the version in the candidate configuration. When the configuration is later committed, only system processes that are affected by the changed configuration elements parse the new configuration.

format—Specifies the format used for the configuration data. There are two acceptable values:

- **text**—Indicates that configuration statements are formatted as ASCII text, using the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements. This is the format used in configuration files stored on the routing platform and displayed by the CLI **show configuration** command.
- **xml**—Indicates that configuration statements are represented by the corresponding Junos XML tag elements. This is the default value if the **format** attribute is omitted.

rescue—Specifies that the rescue configuration replace the current candidate configuration. The only valid value is **rescue**.

rollback—Specifies the numerical index of the previous configuration to load. Valid values are **0** (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is **49**).

url—Specifies the full pathname of the file that contains the configuration data to load. The value can be a local file path, an FTP location, or a Hypertext Transfer Protocol (HTTP) URL:

- A local filename can have one of the following forms:
  - ***/path/filename***—File on a mounted file system, either on the local flash disk or on hard disk.
  - **a:*filename*** or **a:*path/filename***—File on the local drive. The default path is **/** (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.
- A filename on an FTP server has the following form:

  ftp://*username*:*password*@*hostname/path/filename*
- A filename on an HTTP server has the following form:

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

In each case, the default value for the ***path*** variable is the home directory for the username. To specify an absolute path, the application starts the path with the characters **%2F**; for example, **ftp://*username*:*password*@*hostname*/%2F*path/filename***.

Usage Guidelines See "Changing Configuration Information" on page 111.

Related Documentation
- <load-configuration-results> on page 175
- <rpc> on page 179
- replace on page 197
- entries for **<configuration>** and **<configuration-text>** in the *Junos XML API Configuration Reference*

<load-configuration-results>

Usage

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
    <load-error-count>errors</load-error-count>
  </load-configuration-results>
</rpc-reply>
```

Description Enclose one of the two following tag elements, which indicate the status of a configuration loading operation performed by the Junos XML protocol server.

Contents **<load-error-count>**—Specifies the number of errors that occurred when the Junos XML protocol server attempted to load new data into the candidate configuration. The candidate configuration must be restored to a valid state before it is committed.

<load-success/>—Indicates that the Junos XML protocol server successfully loaded new data into the candidate configuration.

Usage Guidelines See "Changing Configuration Information" on page 111.

Related Documentation
- <load-configuration> on page 172
- <rpc-reply> on page 180

<lock-configuration/>

Usage

```
<rpc>
  <lock-configuration/>
</rpc>
```

Description Request that the Junos XML protocol server open and 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 application must emit the **<unlock-configuration/>** tag to unlock the configuration.

If the Junos XML protocol session ends or the application emits the **<unlock-configuration/>** tag before the candidate configuration is committed, all changes made to the candidate are discarded.

Usage Guidelines See "Locking the Candidate Configuration" on page 55.

Related Documentation
- <rpc> on page 179
- <unlock-configuration/> on page 180

<open-configuration>

Usage

```
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>
</rpc>
```

Description Create a private copy of the candidate configuration.

The client application can perform the same operations on the private copy as on the regular candidate configuration, including the commit operation. There are, however, restrictions on the commit operation. For details, see "<commit-configuration>" on page 160.

To discard the private copy without committing it, emit the empty **<close-configuration/>** tag. Changes to the private copy are also lost if the Junos XML protocol session ends for any reason before the changes are committed. It is not possible to save changes to a private copy other than by emitting the **<commit-configuration/>** tag.

Usage Guidelines See "Creating a Private Copy of the Configuration" on page 58.

Related Documentation

- <close-configuration/> on page 160
- <commit-configuration> on page 160
- <lock-configuration/> on page 176
- <rpc> on page 179

<reason>

Usage

```
<xnm:error | xnm:warning>
  <reason>
    <daemon>process</daemon>
    <process-not-configured/>
    <process-disabled/>
    <process-not-running/>
  </reason>
</xnm:error | xnm:warning>
```

Description Explain why a process could not service a request.

Contents

<daemon>—Identifies the process.

<process-disabled>—Indicates that the process has been explicitly disabled by an administrator.

<process-not-configured>—Indicates that the process has been disabled because it is not configured.

<process-not-running>—Indicates that the process is not running.

Usage Guidelines See “Handling an Error or Warning” on page 53.

Related Documentation

- <xnm:error> on page 181
- <xnm:warning> on page 183

<request-end-session/>

Usage

```
<rpc>
  <request-end-session/>
</rpc>
```

Description Request that the Junos XML protocol server end the current session.

Usage Guidelines See “Ending a Junos XML Protocol Session and Closing the Connection” on page 58.

Related Documentation

- <end-session/> on page 166
- <rpc> on page 179

<request-login>

Usage

```
<rpc>
  <request-login>
    <username>account</username>
    <challenge-response>password</challenge-response>
  </request-login>
</rpc>
```

Description Request authentication by the Junos XML protocol server when using the clear-text or SSL access protocol.

Emitting both the **<username>** and **<challenge-response>** tag elements is appropriate if the client application automates access to device information and does not interact with users, or obtains the password from a user before beginning the authentication process.

Emitting only the **<username>** tag element is appropriate if the application does not obtain the password until the authentication process has already begun. In this case, the Junos XML protocol server returns the **<challenge>** tag element to request the password associated with the account.

Contents **<challenge-response>**—Specifies the password for the account named in the **<username>** tag element. Omit this tag element to have the Junos XML protocol server emit the **<challenge>** tag element to request the password.

<username>—Names the account under which to authenticate with the Junos XML protocol server. The account must already be configured on the device where the Junos XML protocol server is running.

Usage Guidelines See “Submitting an Authentication Request” on page 46.

Related Documentation

- <challenge> on page 158
- <rpc> on page 179

<routing-engine>

Usage

```
<rpc-reply xmlns:junos="URL">
  <commit-results>

  <!-- when the candidate configuration is committed -->
    <routing-engine>
      <name>reX</name>
      <commit-success/>
    </routing-engine>

  <!-- when the candidate configuration is syntactically valid -->
    <routing-engine>
      <name>reX</name>
      <commit-check-success/>
```



```
      </routing-engine>

    </commit-results>
  </rpc-reply>
```

Description Enclose tag elements indicating that the Junos XML protocol server successfully fulfilled a commit request.

Contents **<commit-check-success>**—Indicates that the candidate configuration is syntactically correct.

<commit-success>—Indicates that the Junos XML protocol server successfully committed the candidate configuration.

<name>—Name of the Routing Engine on which the commit operation was performed. Possible values are **re0** and **re1**.

Usage Guidelines See "Committing a Configuration" on page 141.

Related Documentation

- <commit-results> on page 164
- <rpc-reply> on page 180

<rpc>

Usage

```
<junoscript>
  <rpc [attributes]>
    <!-- tag elements in a request from a client application -->
  </rpc>
</junoscript>
```

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 Junos XML protocol server echoes the attribute unchanged in its opening **<rpc-reply>** tag, making it simple to map the response to the initiating request.

NOTE: The **xmlns:junos** attribute name is reserved. The Junos XML protocol server sets the attribute to an appropriate value on the opening <rpc-reply> tag, so client applications must not emit it on the opening <rpc> tag.

Usage Guidelines See "Sending a Request to the Junos XML Protocol Server" on page 48.

Related Documentation

- <junoscript> on page 170
- <rpc-reply> on page 180

<rpc-reply>

Usage

```
<junoscript>
<rpc-reply xmlns:junos="namespace-URL">
    <!-- tag elements in a reply from the Junos XML protocol server -->
</rpc-reply>
</junoscript>
```

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

- The tag element used to enclose data generated by the Junos XML protocol server or a Junos OS module in response to a client application's request.
- The **<output>** tag element, if the Junos XML API does not define a specific tag element for requested information.

Attributes **xmlns:junos**—Names the XML namespace for the Junos XML tag elements enclosed by the **<rpc-reply>** tag element that have the **junos:** prefix. The value is a URL of the form **http://xml.juniper.net/junos/*release-code*/junos**, where ***release-code*** is the standard string that represents a release of the Junos OS, such as **10.4R1** for the initial version of Junos OS Release 10.4.

Usage Guidelines See "Parsing the Junos XML Protocol Server Response" on page 51.

Related Documentation

- <junoscript> on page 170
- **<output>** in the *Junos XML API Operational Reference*
- <rpc> on page 179

<unlock-configuration/>

Usage

```
<rpc>
  <unlock-configuration/>
</rpc>
```

Description Request that the Junos XML protocol server unlock and close the candidate configuration. Until the application emits this tag, other users or applications can read the configuration but cannot change it.

Usage Guidelines See "Unlocking the Candidate Configuration" on page 56.

Related Documentation

- <lock-configuration/> on page 176

- <rpc> on page 179

<?xml?>

Usage

```
<?xml version="version" encoding="encoding"?>
```

Description Specify the XML version and character encoding scheme for the session.

Attributes **encoding**—Specifies the standardized character set that the emitter uses and can interpret.

version—Specifies the version of XML used by the emitter.

Usage Guidelines See "Emitting the <?xml?> PI" on page 41 and "Parsing the Junos XML Protocol Server's <?xml?> PI" on page 43.

Related Documentation

- <junoscript> on page 170

<xnm:error>

Usage

```
<junoscript>
  <any-child-of-junoscript>
    <xnm:error xmlns="namespace-URL" xmlns:xnm="namespace-URL">
      <parse/>
      <source-daemon>module-name </source-daemon>
      <filename>filename</filename>
      <line-number>line-number </line-number>
      <column>column-number</column>
      <token>input-token-id </token>
      <edit-path>edit-path</edit-path>
      <statement>statement-name </statement>
      <message>error-string</message>
      <re-name>re-name-string</re-name>
      <database-status-information>...</database-status-information>
      <reason>...</reason>
    </xnm:error>
  </any-child-of-junoscript>
</junoscript>
```

Description Indicate that the Junos XML protocol server has experienced an error while processing the client application's request. If the server has already emitted the response tag element for the current request, the information enclosed in the response tag element might be incomplete. The client application must include code that discards or retains the information, as appropriate. The child tag elements described in the Contents section detail the nature of the error. The Junos XML protocol server does not necessarily emit all child tag elements; it omits tag elements that are not relevant to the current request.

Attributes

xmlns—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Contents

<column>—(Occurs only during loading of a configuration file) Identifies the element that caused the error by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are specified by the accompanying **<line-number>** and **<filename>** tag elements.

<edit-path>—(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the error occurred, in the form of the CLI configuration mode banner.

<filename>—(Occurs only during loading of a configuration file) Names the configuration file that was being loaded.

<line-number>—(Occurs only during loading of a configuration file) Specifies the line number where the error occurred in the configuration file that was being loaded, which is named by the accompanying **<filename>** tag element.

<message>—Describes the error in a natural-language text string.

<parse/>—Indicates that there was a syntactic error in the request submitted by the client application.

<re-name>—Names the Routing Engine on which the error occurred.

<source-daemon>—Names the Junos OS module that was processing the request in which the error occurred.

<statement>—(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The accompanying **<edit-path>** tag element specifies the statement's parent hierarchy level.

<token>—Names which element in the request caused the error.

The other tag elements are explained separately.

Usage Guidelines

See "Handling an Error or Warning" on page 53.

Related Documentation

- <database-status-information> on page 166
- <junoscript> on page 170
- <reason> on page 177
- <xnm:warning> on page 183

<xnm:warning>

Usage

```
<junoscript>
  <any-child-of-junoscript>
    <xnm:warning xmlns="namespace-URL" xmlns:xnm="namespace-URL">
      <source-daemon>module-name </source-daemon>
      <filename>filename</filename>
      <line-number>line-number </line-number>
      <column>column-number</column>
      <token>input-token-id </token>
      <edit-path>edit-path</edit-path>
      <statement>statement-name </statement>
      <message>error-string</message>
      <reason>...</reason>
    </xnm:warning>
  </any-child-of-junoscript>
</junoscript>
```

Description Indicate that the server has encountered a problem while processing the client application's request. The child tag elements described in the Contents section detail the nature of the warning.

Attributes **xmlns**—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix in their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Contents **<column>**—(Occurs only during loading of a configuration file) Identifies the element that caused the problem by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are specified by the accompanying **<line-number>** and **<filename>** tag elements.

<edit-path>—(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the problem occurred, in the form of the CLI configuration mode banner.

<filename>—(Occurs only during loading of a configuration file) Names the configuration file that was being loaded.

<line-number>—(Occurs only during loading of a configuration file) Specifies the line number where the problem occurred in the configuration file that was being loaded, which is named by the accompanying **<filename>** tag element.

<message>—Describes the warning in a natural-language text string.

<source-daemon>—Names the Junos OS module that was processing the request in which the warning occurred.

<statement>—(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The accompanying **<edit-path>** tag element specifies the statement's parent hierarchy level.

<token>—Names which element in the request caused the warning.

The other tag element is explained separately.

Usage Guidelines See "Handling an Error or Warning" on page 53.

Related Documentation

- <junoscript> on page 170
- <reason> on page 177
- <xnm:error> on page 181

CHAPTER 8

Summary of Attributes in Junos XML Tags

This chapter lists the attributes that client applications include in an opening Junos XML tag when performing some operations on configuration elements, such as deletion, renaming, and reordering. It also lists the attributes that the Junos XML protocol server includes in an opening XML tag when returning certain kinds of information. The entries are in alphabetical order. For information about the notational conventions used in this chapter, see Table 2 on page xxii.

active

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->
        <element active="active">
          <name>identifier</name> <!-- if element has an identifier -->
        </element>
      <!-- closing tag for each parent of the element -->
    </configuration>
  </load-configuration>
</rpc>
```

Description Reactivate a previously deactivated configuration element.

The **active** attribute can be combined with one or more of the **insert**, **rename**, or **replace** attributes. To deactivate an element, include the **inactive** attribute instead.

Usage Guidelines See "Changing a Configuration Element's Activation State" on page 133 and "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135.

Related Documentation

- inactive on page 187
- insert on page 188
- <load-configuration> on page 172
- rename on page 196
- replace on page 197
- <rpc> on page 179

count

Usage

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
        <object-type count="count"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

Description Specify the number of configuration objects of the specified type about which to return information. If the attribute is omitted, the Junos XML protocol server returns information about all objects of the type.

The attribute can be combined with one or more of the **matching**, **recurse**, and **start** attributes.

If the application requests Junos XML-tagged output (the default), the Junos XML protocol server includes two attributes in the opening container tag for each returned object:

- **junos:position**—Specifies the object's numerical index.
- **junos:total**—Reports the total number of such objects that exist in the hierarchy.

These attributes do not appear if the application requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

Usage Guidelines See "Requesting a Specified Number of Configuration Objects" on page 93.

Related Documentation

- <get-configuration> on page 167
- matching on page 195
- recurse on page 196
- <rpc> on page 179
- start on page 198

delete

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->

      <!-- For a hierarchy level or object without an identifier -->
        <level-or-object delete="delete">

      <!-- For an object with an identifier (here, called <name>) -->
```



```
      <object delete="delete">
        <name>identifier</name>
      </object>

    <!-- For a single-value or fixed-form option of an object -->
      <object>
        <name>identifier</name> <!-- if the object has an identifier -->
        <option delete="delete"/>
      </object>

    <!-- closing tag for each parent of the element -->

    <!-- For a value in a multivalued option of an object -->
    <!-- opening tag for each parent of the parent object -->
        <parent-object>
          <name>identifier</name>
        <object delete="delete">value</object>
      </parent-object>
  <!-- closing tag for each parent of the parent object -->

    </configuration>
  </load-configuration>
</rpc>
```

Description Specify that the Junos XML protocol server remove the configuration element from the candidate configuration. The only acceptable value for the attribute is **delete**.

Usage Guidelines See "Deleting Configuration Elements" on page 124.

Related Documentation

- <load-configuration> on page 172
- <rpc> on page 179

inactive

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->

        <!-- if immediately deactivating a newly created element -->
        <element inactive="inactive">
          <name>identifier</name> <!-- if element has an identifier -->
          <!-- tag elements for each child of the element -->
        </element>

        <!-- if deactivating an existing element -->
        <element inactive="inactive">
          <name>identifier</name> <!-- if element has an identifier -->
        </element>
      <!-- closing tag for each parent of the element -->
    </configuration>
  </load-configuration>
```

```
</rpc>
```

Description Deactivate a configuration element. The element remains in the candidate configuration or private copy, but when the configuration is later committed, the element does not affect the functioning of the routing, switching, or security platform.

The **inactive** attribute can be combined with one or more of the **insert**, **rename**, or **replace** attributes, as described in "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135. To reactivate a deactivated element, include the **active** attribute instead.

Usage Guidelines See "Changing a Configuration Element's Activation State" on page 133.

Related Documentation

- active on page 185
- insert on page 188
- <load-configuration> on page 172
- rename on page 196
- <rpc> on page 179

insert

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the set -->

        <!-- if each element in the ordered set has one identifier -->
        <ordered-set insert="(before | after)" name="referent-value">
          <name>value-for-moving-object</name>
        </ordered-set>

        <!-- if each element in the ordered set has two identifiers -->
        <ordered-set insert="(before | after)" identifier1="referent-value" \
              identifier2="referent-value">
          <identifier1>value-for-moving-object</identifier1>
          <identifier2>value-for-moving-object</identifier2>
        </ordered-set>

      <!-- closing tag for each parent of the set -->
    </configuration>
  </load-configuration>
</rpc>
```

Description Change the position of a configuration element in an ordered set. The new position is specified relative to a reference element, which is specified by including an attribute named after each of its identifier tags. In the Usage section, the identifier tag element is called **<name>** when each element in the set has one identifier.

The **insert** attribute can be combined with either the **active** or **inactive** attribute, as described in "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135.

Usage Guidelines See "Reordering Elements in Configuration Objects" on page 129.

Related Documentation

- active on page 185
- inactive on page 187
- <load-configuration> on page 172
- <rpc> on page 179

junos:changed

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration standard-attributes junos:changed="changed">
    <!-- opening-tag-for-each-parent-level junos:changed="changed" -->

      <!-- If the changed element is an empty tag -->
      <element junos:changed="changed"/>

      <!-- If the changed element has child tag elements -->
      <element junos:changed="changed">
        <first-child-of-element junos:changed="changed">
        <second-child-of-element junos:changed="changed">
        <!-- additional children of element - ->
      </element>

    <!-- closing-tag-for-each-parent-level -->
  </configuration>
</rpc-reply>
```

Description Indicate that a configuration element has changed since the last commit operation. The Junos XML protocol server includes the attribute when the client application includes the **changed** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag. The attribute appears in the opening tag of every parent tag element in the path to the changed configuration element, including the opening top-level **<configuration>** tag.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

For information about the standard attributes in the opening **<configuration>** tag, see "Requesting Information from the Committed or Candidate Configuration" on page 70.

Usage Guidelines See "Requesting a Change Indicator for Configuration Elements" on page 76.

Related Documentation
- <get-configuration> on page 167
- <rpc-reply> on page 180

junos:changed-localtime

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration 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 70.

Related Documentation
- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 180
- junos:changed-seconds on page 190

junos:changed-seconds

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration 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 70.

Related Documentation
- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 180
- junos:changed-localtime on page 190

junos:commit-localtime

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration 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 70.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 180
- junos:commit-user on page 191
- junos:commit-seconds on page 191

junos:commit-seconds

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration 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 number of seconds since midnight on 1 January 1970.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 70.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 180
- junos:commit-user on page 191
- junos:commit-localtime on page 190

junos:commit-user

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration 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 Junos OS username of the user who requested the commit operation.

Usage Guidelines See "Requesting Information from the Committed or Candidate Configuration" on page 70.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Reference*
- <rpc-reply> on page 180
- junos:commit-localtime on page 190
- junos:commit-seconds on page 191

junos:group

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tag for each parent of the element -->
      <inherited-element junos:group="source-group">
        <inherited-child-of-inherited-element junos:group="source-group">
          <!-- inherited-children-of-child junos:group="source-group" -->
        </inherited-child-of-inherited-element>
      </inherited-element>
    <!-- closing tag for each parent of the element  -->
  </configuration>
</rpc-reply>
```

Description Name the configuration group from which each configuration element is inherited. The Junos XML protocol server includes the attribute when the client application includes the **groups** and **inherit** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag. Instead, the Junos XML protocol server provides the information in a comment directly above each inherited element.

Usage Guidelines See "Displaying the Source Group for Inherited Configuration Elements" on page 82.

Related Documentation

- <get-configuration> on page 167
- <rpc-reply> on page 180

junos:interface-range

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <interfaces>
      <!-- For each inherited element -->
      <interface junos:interface-range="source-interface-range">
```



```
        <inherited-element junos:interface-range="source-interface-range">
          <inherited-child-of-inherited-element
                junos:interface-range="source-interface-range">
            <!-- inherited-children-of-child
                  junos:interface-range="source-interface-range" -->
          </inherited-child-of-inherited-element>
        </inherited-element>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>
```

Description Name the interface range from which each configuration element is inherited. The Junos XML protocol server includes the attribute when the client application includes the **interface-ranges** and **inherit** attributes in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

Usage Guidelines See "Displaying the Source Interface Range for Inherited Configuration Elements" on page 87.

Related Documentation

- <get-configuration> on page 167
- <rpc-reply> on page 180

junos:key

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tag for each parent of the object -->
      <object>
        <name junos:key="key">identifier</name>
        <!-- additional children of object -->
      </object>
    <!-- closing tag for each parent of the object  -->
  </configuration>
</rpc-reply>
```

Description Indicate that a child configuration tag element is the identifier for its parent tag element. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the **<get-configuration>** tag element) and has included the **junos:key** attribute in the opening **<junoscript>** tag for the current session.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

Usage Guidelines See "Requesting an Indicator for Identifiers" on page 74.

Related Documentation

- <get-configuration> on page 167
- <junoscript> on page 170
- <rpc> on page 179

junos:position

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tags for each parent of the object -->
      <object junos:position="index" junos:total="total" >
    <!-- closing tags for each parent of the object -->
  </configuration>
</rpc-reply>
```

Description Specify the index number of the configuration object in the list of objects of a specified type about which information is being returned. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the **<get-configuration>** tag element) and includes the **count** attribute, the **start** attribute, or both, in the opening tag for the object type.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

Usage Guidelines See “Requesting a Specified Number of Configuration Objects” on page 93.

Related Documentation

- count on page 186
- <get-configuration> on page 167
- junos:total on page 194
- <rpc> on page 179
- start on page 198

junos:total

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tags for each parent of the object -->
      <object junos:position="index" junos:total="total">
    <!-- closing tags for each parent of the object -->
  </configuration>
</rpc-reply>
```

Description Specify the number of configuration objects of a specified type about which information is being returned. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the **<get-configuration>** tag

element) and includes the **count** attribute, the **start** attribute, or both, in the opening tag for the object type.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

Usage Guidelines See "Requesting a Specified Number of Configuration Objects" on page 93.

Related Documentation

- count on page 186
- <get-configuration> on page 167
- junos:position on page 194
- <rpc> on page 179
- start on page 198

matching

Usage

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
        <level matching="matching-expression"/>
      <!-- closing tags for each parent of the level -->
  </configuration>
  </get-configuration>
</rpc>
```

Description Request information about only those instances of a configuration object type at the specified level in the configuration hierarchy that have the specified set of characters in their identifier names (characters that match a regular expression). If the attribute is omitted, the Junos XML protocol server returns the complete set of child tag elements for the specified parent level.

The attribute can be combined with one or more of the **count**, **recurse**, and **start** attributes.

To represent the objects to return, the ***matching-expression*** value uses a slash-separated list of hierarchy level and object names similar to an XML Path Language (XPath) representation. Each level in the representation can be either a full level name or a regular expression that matches the identifier name of one or more instances of an object type:

object-type[name='*regular-expression*']"

The regular expression uses the notation defined in POSIX Standard 1003.2 for extended (modern) UNIX regular expressions. For details about the notation, see "Requesting a Subset of Objects by Using Regular Expressions" on page 99.

Usage Guidelines See "Requesting a Subset of Objects by Using Regular Expressions" on page 99.

Related Documentation

- count on page 186
- <get-configuration> on page 167
- <rpc> on page 179
- start on page 198

recurse

Usage

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
        <object-type recurse="false"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

Description Request only the identifier tag element for each configuration object of a specified type in the configuration hierarchy. If the attribute is omitted, the Junos XML protocol server returns the complete set of child tag elements for every object. The only acceptable value for the attribute is **false**.

The attribute can be combined with one or more of the **count**, **matching**, and **start** attributes.

Usage Guidelines See “Requesting Identifiers Only” on page 95.

Related Documentation

- count on page 186
- <get-configuration> on page 167
- <rpc> on page 179
- start on page 198

rename

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the object -->

        <!-- if the object has one identifier -->
        <object rename="rename" name="new-name">
          <name>current-name</name>
        </object>

        <!-- if the object has two identifiers, both changing -->
        <object rename="rename" identifier1="new-name" \
```



```
        identifier2=new-name">
      <identifier1>current-name</identifier1>
      <identifier2>current-name</identifier2>
    </object>

    <!-- closing tag for each parent of the object -->
  </configuration>
 </load-configuration>
</rpc>
```

Description Change the name of one or more of a configuration object's identifiers. In the Usage section, the identifier tag element is called **<name>** when the element has one identifier.

The **rename** attribute can be combined with either the **inactive** or **active** attribute.

Usage Guidelines See "Renaming a Configuration Object" on page 131.

Related Documentation

- active on page 185
- inactive on page 187
- <load-configuration> on page 172
- <rpc> on page 179

replace

Usage

```
<rpc>
  <load-configuration action="replace">
    <configuration>
      <!-- opening tag for each parent of the element -->
        <container-tag replace="replace">
          <name>identifier</name>
          <!-- tag elements for other children, if any -->
        </container-tag>
      <!-- closing tag for each parent of the element -->
    </configuration>
  </load-configuration>
</rpc>
```

Description Specify that the configuration element completely replace the element in the candidate configuration that has the same identifier (in the Usage section, the identifier tag element is called **<name>**). If the attribute is omitted, the Junos XML protocol server merges the element with the existing element as described in "Merging Configuration Elements" on page 118. The only acceptable value for the attribute is **replace**.

The client application must also include the **action="replace"** attribute in the opening **<load-configuration>** tag.

The **replace** attribute can be combined with either the **active** or **inactive** attribute, as described in "Changing a Configuration Element's Activation State Simultaneously with Other Changes" on page 135.

Usage Guidelines See "Replacing Configuration Elements" on page 121.

Related Documentation
- active on page 185
- inactive on page 187
- <load-configuration> on page 172
- <rpc> on page 179

start

Usage

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
        <object-type start="index"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

Description Specify the index number of the first object to return (**1** for the first object, **2** for the second, and so on) when requesting information about a configuration object of a specified type. If the attribute is omitted, the returned set of objects starts with the first one in the configuration hierarchy.

The attribute can be combined with one or more of the **count**, **matching**, and **recurse** attributes.

If the application requests Junos XML-tagged output (the default), the Junos XML protocol server includes two attributes in the opening container tag for each returned object:

- **junos:position**—Specifies the object's numerical index.
- **junos:total**—Reports the total number of such objects that exist in the hierarchy.

These attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

Usage Guidelines See "Requesting a Specified Number of Configuration Objects" on page 93.

Related Documentation
- count on page 186
- <get-configuration> on page 167
- recurse on page 196

- <rpc> on page 179

xmlns

Usage

```
<rpc-reply xmlns:junos="URL">
  <operational-response xmlns="URL-for-DTD">
    <!-- Junos XML tag elements for the requested information -->
  </operational-response>
</rpc-reply>
```

Description 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.

Usage Guidelines See "Requesting Operational Information" on page 66.

Related Documentation

- <rpc-reply> on page 180

PART 3

Writing Junos XML Protocol Client Applications

CHAPTER 9

Writing Junos XML Protocol Perl Client Applications

Juniper Networks provides a Perl module **JUNOS::Device** to help you more quickly and easily develop custom Perl scripts for configuring and monitoring switches, routers, and security devices running Junos OS. The module implements a **JUNOS::Device** object that client applications can use to communicate with the Junos XML protocol server on a device running Junos OS. 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 Junos::Device Perl Module and Sample Scripts

The Junos XML protocol 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 **JUNOS** module and its supporting files, and an **examples** directory for the sample scripts.

Client applications use the **JUNOS::Device** object to communicate with a Junos XML protocol server. The library contains several modules, but client applications directly invoke only the **JUNOS::Device** object. 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 Junos 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** method 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 Junos XML protocol Perl distribution.
- load_configuration.pl—Illustrates how to change a device configuration by loading a file that contains configuration data formatted with Junos XML tag elements. The distribution includes two sample configuration files, set_login_class_bar.xml and set_login_user_foo.xml; however, you can specify another configuration file on the command line. The script is provided in the examples/load_configuration directory in the Junos XML protocol Perl distribution.

The following sample scripts are used together to illustrate how to store and retrieve data from the Junos XML API (or any XML-tagged data set) in a relational database. Although these scripts create and manipulate MySQL tables, the data manipulation techniques that they illustrate apply to any relational database. The scripts are provided in the **examples/RDB** directory in the Perl distribution:

- **get_config.pl**—Illustrates how to retrieve routing platform configuration information.
- **make_tables.pl**—Generates a set of Structured Query Language (SQL) statements for creating relational database tables.
- **pop_tables.pl**—Populates existing relational database tables with data extracted from a specified XML file.
- **unpop_tables.pl**—Transforms data stored in a relational database table into XML and writes it to a file.

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

Downloading the Junos XML Protocol Perl Client and Prerequisites Package

To download the compressed tar archive files that contain the Junos XML protocol Perl client distribution and the prerequisites package, perform the following steps:

1. Access the Junos XML protocol download page on the Juniper Networks website at https://www.juniper.net/support/products/junoscript/ .
2. Click the link for the appropriate software release.
3. Select the software tab.
4. Click the links to download the client distribution and the prerequisites package that support the appropriate access protocols. Customers in the United States and Canada can download the packages that support all access protocols including SSH, SSL, clear-text and Telnet protocols (the domestic package). Customers in other countries can download the packages that support only the clear-text and Telnet protocols (the export package).

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

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 https://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.

Unpacking the Junos XML Protocol Perl Client and Sample Scripts

To uncompress and unpack the gzip tar archive that contains the Junos XML protocol Perl client and sample scripts, perform the following steps:

1. Create the directory where you want to store the Junos XML protocol Perl client application and sample scripts, and move the downloaded files into this directory. Then make it the working directory:

   % mkdir *parent-directory*
   % mv junoscript-perl-*release-type*.tar.gz *parent-directory*
   % cd *parent-directory*

2. Issue the following command to uncompress and unpack the Junos XML protocol Perl client package:

   - On FreeBSD and Linux systems:

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

   - On Solaris systems:

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

where ***release*** is the release code (e.g. **10.4R1.1**) and ***type*** is **domestic** or **export**.

Step 2 creates a directory called **junoscript-perl-*release*** and writes the contents of the package to it. A typical filename for the compressed tar archive might be junoscript-perl-9.5R1.8-domestic.tar.gz. Unpacking this archive creates the directory junoscript-perl-9.5R1.8/ directly under the *parent-directory* directory and moves the application files and sample scripts into this new directory.

Installing the Prerequisites Package and the Junos XML Protocol Perl Client

Perl must be installed on your system before you install the prerequisites package or the client software. The Junos XML protocol Perl client requires Perl version 5.0004 or later.

To confirm whether Perl is installed on your system and to determine which version of Perl is currently running, issue the following commands:

```
% which perl
% perl -v
```

If the issued output indicates that Perl is not installed or the version is older than the required version, you need to download and install the Perl source package located at http://www.cpan.org/src/stable.tar.gz .

After installing a suitable version of Perl, unpack and install the prerequisites package. Then install the Junos XML protocol Perl client application. These procedures are detailed in the following sections:

- Unpacking and Installing the Junos XML Protocol Perl Client Prerequisites Package on page 206
- Installing the Junos XML Protocol Perl Client on page 207

For additional information, consult the **README** file that is included with the client distribution. It is located in the ***parent-directory/*junoscript-perl-*release/*** directory.

Unpacking and Installing the Junos XML Protocol Perl Client Prerequisites Package

The prerequisites package consists of C libraries, executables, and Perl modules. It must be installed on the client machine for the Junos XML protocol Perl client and the included examples to work correctly. To uncompress and unpack the gzip tar archive containing the prerequisite files, perform the following steps:

1. Move the downloaded prerequisites package into the *parent-directory/*junoscript-perl-*release/* directory that was created in Step 2 of "Unpacking the Junos XML Protocol Perl Client and Sample Scripts" on page 205. The gzip tar archive containing the prerequisite files must be uncompressed, unpacked, and installed in this directory.
2. Issue the following command to uncompress and unpack the package:
   - On FreeBSD and Linux systems:

     % tar zxf junoscript-perl-prereqs-*release-type*.tar.gz
   - On Solaris systems:

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

where ***release*** is the release code (e.g. **10.4R1.1**) and ***type*** is **domestic** or **export**. This command creates a directory called prereqs/ and writes the contents of the package to it.

By default, the prerequisite Perl modules are installed in the standard directory. The standard directory is normally /usr/local/lib/. You need root privileges to access the standard directory. You can opt to install the modules in a private directory.

- To install the required modules in the standard directory:
  1. Go to the *parent-directory/*junoscript-perl-*release* directory where you unpacked the prerequisites package.

2. Issue the following command:

   % perl install-prereqs.pl -used_by *example* -force

   where the **-used_by *example*** option is invoked to install only modules used by a specific example, and the **-force** option installs the module even if an older version exists or if the **make test** command fails.

- To install the required modules in a private directory:

1. Set the **PERL5LIB**, **MANPATH**, and **PATH** environment variables.

   % setenv PERL5LIB *private-directory-path*
   % setenv MANPATH "$MANPATH/:$PERL5LIB/../man"
   % setenv PATH "$PATH/:$PERL5LIB/../bin"

   For sh, ksh, and bash shells, **$PERL5LIB** can be set with **EXPORT PERL5LIB=*private-directory-path***

2. Go to the *parent-directory/*junoscript-perl-*release* directory where the prerequisites package was unpacked:

3. Issue the following command:

   % perl install-prereqs.pl -used_by *example* -install_directory $PERL5LIB -force

   where the **-used_by *example*** option is invoked to install only modules used by a specific example, and the **-force** option installs the module even if an older version exists or if the **make test** command fails. The **-install_directory $PERL5LIB** option installs the prerequisite Perl modules in the private directory that you specified in Step 1.

To view any missing dependencies, issue the following command:

% perl required-mod.pl

This command lists the modules that still require installation.

Installing the Junos XML Protocol Perl Client

After installing the prerequisites package as detailed in “Unpacking and Installing the Junos XML Protocol Perl Client Prerequisites Package” on page 206, install the Junos XML protocol Perl client software. Go to the *parent-directory/*junoscript-perl-*release/* directory that was created in Step 2 in “Unpacking the Junos XML Protocol Perl Client and Sample Scripts” on page 205. Perform the following steps to install the client software:

1. Create the makefile:

   - To install the Perl client in the standard directory (generally /usr/local/lib):

     % perl Makefile.pl

     ```
     Checking if your kit is complete...
     Looks good
     Writing Makefile for junoscript-perl
     ```

   - To install the Perl client in a private directory:

Make sure that the **PERL5LIB**, **MANPATH**, and **PATH** environment variables are set as detailed in "Unpacking and Installing the Junos XML Protocol Perl Client Prerequisites Package" on page 206. Then create the makefile:

```
% perl Makefile.PL LIB=$PERL5LIB INSTALLMAN3DIR=$PERL5LIB/../man/man3
```

2. Test and install the application:

```
% make
% make test
% make install
```

The Junos XML protocol Perl client application is installed and ready for use. For information about the **JUNOS::Device** object and a list of valid queries, consult the man page by invoking the **man** command for the **JUNOS::Device** object:

```
% man JUNOS::Device
```

The sample scripts reside in parent-directory/junoscript-perl-*release*/examples/. You can review and run these examples to acquire some familiarity with the client before writing your own applications.

Tutorial: Writing Perl Client Applications

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

Import Perl Modules and Declare Constants

Include the following statements at the start of the application. The first statement imports the functions provided by the **JUNOS::Device** object, which the application uses to connect to the Junos XML protocol server on a device. The second statement provides error checking and enforces Perl coding practices such as declaration of variables before use.

```
use JUNOS::Device;
use strict;
```

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:

- **File::Basename**—Includes functions for processing filenames.
- **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 statements to declare constants for formatting output:

```
use constant OUTPUT_FORMAT => "%-20s%-8s%-8s%-11s%-14s%s\n";
use constant OUTPUT_TITLE =>
     "\n=============== BGP PROBLEM SUMMARY ===============\n\n";
use constant OUTPUT_ENDING =>
     "\n====================================================\n\n";
```

The load_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 Junos XML Protocol Server

The following sections explain how to use the **JUNOS::Device** object to connect to the Junos XML protocol server on a device running Junos OS:

Satisfying Protocol Prerequisites

The Junos XML protocol server supports several access protocols, listed in “Supported Access Protocols” on page 29. For each connection to the Junos XML protocol server on a device, the application must specify the protocol it is using. Using SSH or Secure Sockets Layer (SSL) is recommended because they provide greater security by encrypting all information before transmission across the network.

Before your application can run, you must satisfy the prerequisites for the protocol it uses. For some protocols this involves activating configuration statements on the device, creating encryption keys, or installing additional software on the device running Junos OS or the machine where the application runs. For instructions, see “Prerequisites for Establishing a Connection” on page 29.

Group Requests

Establishing a connection to the Junos XML protocol server on a device is one of the more time- and resource-intensive functions performed by an application. If the application sends multiple requests to a routing platform, 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 Junos XML protocol server at a time. This is because the **JUNOS::Device** object does not return control to the application until it receives the closing </rpc-reply> tag that represents the end of the Junos XML protocol server's response to the current request.

Obtain and Record Parameters Required by the JUNOS::Device Object

The **JUNOS::Device** object takes the following required parameters, specified as keys in a Perl hash:

- The access protocol to use when communicating with the Junos XML protocol server (key name: **access**). For a list of the acceptable values, see "Supported Access Protocols" on page 29. Before the application runs, satisfy the protocol-specific prerequisites described in "Prerequisites for Establishing a Connection" on page 29.
- 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 Junos XML protocol 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: Collecting Parameters Interactively

Each sample script obtains the parameters required by the **JUNOS::Device** 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:dx: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. It can be clear-text, ssl, ssh or telnet.
          Default: telnet.
    -x <format>     The name of the XSL file to display the response.
          Default: xsl/chassis_inventory_csv.xsl
    -o <filename>   File to which to write output, instead of standard output.
    -d          Turn on debugging.\n\n";

    die $usage;
  }
```

The get_chassis_inventory.pl script includes the following code to obtain values from the command line for the four parameters required by the **JUNOS::Device** 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
my $access = $opt{m} || "telnet";
use constant VALID_ACCESSES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESSES =~ /$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 or to the value **telnet** if the **-m** option is not provided. If the specified value does not match one of the values defined by the **VALID_ACCESSES** constant, the script invokes the **output_usage** subroutine to print the usage message.

```
my $access = $opt{m} || "telnet";
use constant VALID_ACCESSES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESSES =~ /$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";
}
```

Obtaining Application-Specific Parameters

In addition to the parameters required by the **JUNOS::Device** object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the Junos XML protocol 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 **JUNOS::Device** 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 **JUNOS::Device** object (discussed in "Obtain and Record Parameters Required by the JUNOS::Device Object" on page 210). Several examples follow.

The following line enables a debugging trace if the user includes the **-d** command-line option. It invokes the **JUNOS::Trace::init** routine defined in the **JUNOS::Trace** module, which is already imported with the **JUNOS::Device** object.

```
JUNOS::Trace::init(1) if $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 Junos XML protocol 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 Junos XML protocol 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.

```
# Retrieve the XSLT file, default is parsed by perl
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}
```

The following code from the load_configuration.pl script defines whether to merge, replace, update, or overwrite the new configuration data into the configuration database (for more information about these operations, see "Changing Configuration Information" on page 111). The first two lines set the **$load_action** variable to the value of the **-a** command-line option, or to the default value **merge** if the option is not provided. If the specified value does not match one of the valid actions defined in the third line, the script invokes the **output_usage** subroutine.

```
# The default action for load_configuration is 'merge'
my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
```

```
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless (VALID_ACTIONS =~ /$load_action/);
```

Converting Disallowed Characters

Scripts that handle configuration data usually accept and output the data either as Junos XML tag elements or as formatted ASCII statements like those used in the Junos OS CLI. As described in "XML and Junos XML Management Protocol Conventions Overview" on page 11, certain characters cannot appear in their regular form in an XML document. These characters include the apostrophe (**'**), the ampersand (**&**), the greater-than (**>**) and less-than (**<**) symbols, and the quotation mark (**"**). Because these characters might appear in formatted ASCII configuration statements, the script must convert the characters to the corresponding predefined entity references.

The load_configuration.pl script uses the **get_escaped_text** subroutine to substitute predefined entity references for disallowed characters (the get_configuration.pl script includes similar code). The script first defines the mappings between the disallowed characters and predefined entity references, and sets the variable **$char_class** to a regular expression that contains all of the entity references, as follows:

```
my %escape_symbols = (
  qq(")  => '&quot;',
  qq(>)  => '&gt;',
  qq(<)  => '&lt;',
  qq(')  => '&apos;',
  qq(&)  => '&amp;'
);
my $char_class = join ("|", map { "($_)" } keys %escape_symbols);
```

The following code defines the **get_escaped_text** subroutine for the load_configuration.pl script. A detailed discussion of the subsections in the routine follows the complete code sample.

```
sub get_escaped_text
{
  my $input_file = shift;
  my $input_string = "";

  open(FH, $input_file) or return undef;

  while(<FH>) {
    my $line = $_;
    $line =~ s/<configuration-text>//g;
    $line =~ s/<\/configuration-text>//g;
    $line =~ s/($char_class)/$escape_symbols{$1}/ge;
    $input_string .= $line;
  }

  return "<configuration-text>$input_string</configuration-text>";
}
```

The first subsection of the preceding code sample reads in a file containing formatted ASCII configuration statements:

```
sub get_escaped_text
{
  my $input_file = shift;
```



```
    my $input_string = "";

    open(FH, $input_file) or return undef;
```

In the next subsection, the subroutine temporarily discards the lines that contain the opening **<get-configuration>** and closing **</get-configuration>** tags, then replaces the disallowed characters on each remaining line with predefined entity references and appends the line to the **$input_string** variable:

```
  while(<FH>) {
    my $line = $_;
    $line =~ s/<configuration-text>//g;
    $line =~ s/<\/configuration-text>//g;
    $line =~ s/($char_class)/$escape_symbols{$1}/ge;
    $input_string .= $line;
  }
```

The subroutine concludes by replacing the opening **<get-configuration>** and closing **</get-configuration>** tags, and returning the converted set of statements:

```
    return "<configuration-text>$input_string</configuration-text>";
  }
```

Establishing the Connection

After obtaining values for the parameters required for the **JUNOS::Device** object (see "Obtain and Record Parameters Required by the JUNOS::Device Object" on page 210), each sample script records them in the **%deviceinfo** hash:

```
  my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
  );
```

The script then invokes the Junos XML protocol-specific **new** subroutine to create a **JUNOS::Device** 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 JUNOS::Device(%deviceinfo);
  unless ( ref $jnx ) {
    die "ERROR: $deviceinfo{hostname}: failed to connect.\n";
```

Submitting a Request to the Junos XML Protocol Server

After establishing a connection to a Junos XML protocol server (see "Establishing the Connection" on page 215), your application can submit one or more requests by invoking the Perl methods that are supported in the version of the Junos 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 "Mapping CLI Commands to Perl Methods" on page 230 and the files stored in the **lib/JUNOS/*release*** directory of the Junos XML protocol Perl distribution (***release*** is the Junos OS version code, such as

10.4R1 for the initial version of Junos OS Release 10.4). 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/JUNOS/Methods.pm file in the Junos XML protocol Perl distribution. For more information about configuration operations, see "Changing Configuration Information" on page 111 and "Summary of Junos XML Protocol Tag Elements" on page 157.

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/JUNOS/Methods.pm and lib/JUNOS/*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_system_uptime_information** method:

  ```
  ## Method : <get-system-uptime-information>
  ## Returns: <system-uptime-information>
  ## Command: "show system uptime"
  get_system_uptime_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_system_uptime_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-software-information>
  ## Returns: <software-information>
  ## Command: "show version"
  get_software_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_software_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-cos-drop-profile-information>
  ## Returns: <cos-drop-profile-information>
  ## Command: "show class-of-service drop-profile"
  ```



```
get_cos_drop_profile_information => {
   profile_name => $STRING,
},
```

To include a variable value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_cos_drop_profile_information(profile_name => 'user-drop-profile');
```

- An attribute is defined as type **$ATTRIBUTE**. In the following example, the **load_configuration** method takes the **rollback** attribute:

```
load_configuration => {
  rollback => $ATTRIBUTE
},
```

To include a numerical attribute value when invoking a method, set it to the appropriate value. The following example rolls the candidate configuration back to the previous configuration that has an index of 2:

```
$jnx->load_configuration(rollback => 2);
```

To include a string attribute value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_configuration(format => 'text');
```

- A set of configuration statements or corresponding tag elements is defined as type **$DOM**. In the following example, the **get_configuration** method takes a set of configuration statements (along with two attributes):

```
get_configuration => {
   configuration => $DOM,
   format => $ATTRIBUTE,
   database => $ATTRIBUTE,
},
```

To include a set of configuration statements when invoking a method, provide a parsed set of statements or tag elements. The following example refers to a set of Junos XML configuration tag elements in the config-input.xml file. For further discussion, see "Example: Loading Configuration Statements" on page 220.

```
my $parser = new XML::DOM::Parser;
$jnx->load_configuration(
  format => 'xml',
  action => 'merge',
  configuration => $parser->parsefile(config-input.xml)
);
```

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 Junos XML protocol server and shows how to handle error conditions. The **$jnx** variable is defined to be a **JUNOS::Device** object, as discussed in "Establishing the Connection" on page 215. A detailed discussion of the functional subsections follows the complete code sample.

```
my %arguments = ( );
%arguments = ( argument1  =>  value1 ,
   argument2  =>  value2 , ...);
   argument3  =>  value3 ,
  ...);

my $res = $jnx-> method (%args);

unless ( ref $res ) {
  $jnx->request_end_session( );
  $jnx->disconnect( );
  print "ERROR: Could not send request to $hostname\n";
}

my $err = $res->getFirstError( );
if ($err) {
  $jnx->request_end_session( );
  $jnx->disconnect();
  print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

The first subsection of the preceding code sample creates a hash called **%arguments** to define values for a method's options or attributes. For each argument, the application uses the notation described in "Providing Method Options or Attributes" on page 216.

```
my %arguments = ( );
%arguments = ( argument1  =>  value1 ,
   argument2  =>  value2 , ...);
```



```
    argument3  =>  value3 ,
   ...);
```

The application then invokes the method, defining the **$res** variable to point to the **JUNOS::Response** object that the Junos XML protocol server returns in response to the request (the object is defined in the lib/JUNOS/Response.pm file in the Junos XML protocol Perl distribution):

```
my $res = $jnx-> method (%args);
```

If the attempt to send the request failed, the application prints an error message and closes the connection:

```
unless ( ref $res ) {
  $jnx->request_end_session();
  $jnx->disconnect();
  print "ERROR: Could not send request to $hostname\n";
}
```

If there was an error in the Junos XML protocol server's response, the application prints an error message and closes the connection. The **getFirstError** function is defined in the **JUNOS::Response** module (lib/JUNOS/Response.pm) in the Junos XML protocol Perl distribution.

```
my $err = $res->getFirstError( );
if ($err) {
  $jnx->request_end_session( );
  $jnx->disconnect( );
  print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

Example: Getting 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, switching, or security platform. It is equivalent to issuing the **show chassis hardware detail** command.

After establishing a connection to the Junos XML protocol 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. For more information, see "Parsing and Formatting an Operational Response" on page 224.

```
# send the command and receive a XML::DOM object
my $res = $jnx->$query( %queryargs );
unless ( ref $res ) {
  die "ERROR: $deviceinfo{hostname}: failed to execute command $query.\n";
}
# Check and see if there were any errors in executing the command.
my $err = $res->getFirstError( );
if ($err) {
  print STDERR "ERROR: $deviceinfo{'hostname'} - ", $err->{message}, "\n";
} else {
```

```
    # Now do the transformation using XSLT
    ... code that uses XSLT to process results ...
}
```

Example: Loading Configuration Statements

The load_configuration.pl script loads configuration statements onto a device. It uses the basic structure for sending requests described in "Submitting a Request" on page 218 but also defines a **graceful_shutdown** subroutine that handles errors in a slightly more elaborate manner than that described in "Submitting a Request" on page 218. The following sections describe the different functions that the script performs:

- Handling Error Conditions on page 220
- Locking the Configuration on page 221
- Reading In and Parsing the Configuration Data on page 221
- Loading the Configuration Data on page 222
- Committing the Configuration on page 223

Handling Error Conditions

The **graceful_shutdown** subroutine in the load_configuration.pl script handles errors in a slightly more elaborate manner than the generic structure described in "Submitting a Request" on page 218. It employs the following additional 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 load_configuration.pl script. The **if** statement for **STATE_CONNECTED** is similar to the error checking described in "Submitting a Request" on page 218. The **eval** statement used in each case ensures that any errors that occur during execution of the enclosed function call are trapped so that failure of the function call does not cause the script to exit.

```
sub graceful_shutdown
{
  my ($jnx, $req, $state, $success) = @_;

 if ($state >= STATE_CONFIG_LOADED) {
    print "Rolling back configuration ...\n";
    eval {
      $jnx->load_configuration(rollback => 0);
    };
 }

 if ($state >= STATE_LOCKED) {
    print "Unlocking configuration database ...\n";
    eval {
      $jnx->unlock_configuration();
    };
 }
```



```
  if ($state >= STATE_CONNECTED) {
     print "Disconnecting from the device ...\n";
     eval {
        $jnx->request_end_session()
        $jnx->disconnect();
     };
  }

  if ($success) {
     die "REQUEST $req SUCCEEDED\n";
  } else {
     die "REQUEST $req FAILED\n";
  };
}
```

Locking the Configuration

The main section of the load_configuration.pl script begins by establishing a connection to a Junos XML protocol server, as described in "Establishing the Connection" on page 215. 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 220.

```
print "Locking configuration database ...\n";
my $res = $jnx->lock_configuration();
my $err = $res->getFirstError();
if ($err) {
   print "ERROR: $deviceinfo{hostname}: failed to lock configuration. Reason:
$err->{message}.\n";
   graceful_shutdown($jnx, $xmlfile, STATE_CONNECTED, REPORT_FAILURE);
}
```

Reading In and Parsing the Configuration Data

In the following code sample, the load_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, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

my $parser = new XML::DOM::Parser;
...

my $doc;
if ($opt{t}) {
   my $xmlstring = get_escaped_text($xmlfile);
   $doc = $parser->parsestring($xmlstring) if $xmlstring;

} else {
   $doc = $parser->parsefile($xmlfile);
}
```

```
unless ( ref $doc ) {
  print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is
well-formed\n";
  graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

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, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

If the **-t** command-line option was included when the load_configuration.pl script was invoked, the file referenced by the **$xmlfile** variable should contain formatted ASCII configuration statements like those returned by the CLI configuration-mode **show** command. The script invokes the **get_escaped_text** subroutine described in “Converting Disallowed Characters” on page 214, assigning the result to the **$xmlstring** variable. The script invokes the **parsestring** function to transform the data in the file into the proper format for loading into the configuration hierarchy, and assigns the result to the **$doc** variable. The **parsestring** function is defined in the **XML::DOM::Parser** module, and the first line in the following sample code instantiates the module as an object, setting the **$parser** variable to refer to it:

```
my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
  my $xmlstring = get_escaped_text($xmlfile);
  $doc = $parser->parsestring($xmlstring) if $xmlstring;
```

If the file contains Junos XML configuration tag elements instead, the script invokes the **parsefile** function (also defined in the **XML::DOM::Parser** module) on the file:

```
} else {
  $doc = $parser->parsefile($xmlfile);
}
```

If the parser cannot transform the file, the script invokes the **graceful_shutdown** subroutine described in “Handling Error Conditions” on page 220:

```
unless ( ref $doc ) {
  print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is
well-formed\n";
  graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

Loading the Configuration Data

The script now invokes the **load_configuration** method to load the configuration onto thedevice. It places the statement inside an **eval** block to ensure that the

graceful_shutdown subroutine is invoked if the response from the Junos XML protocol server has errors.

```
eval {
$res = $jnx->load_configuration(
  format => $config_format,
  action => $load_action,
  configuration => $doc);
};
if ($@) {
  print "ERROR: Failed to load the configuration from $xmlfile.  Reason: $@\n";
  graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
  exit(1);
}
```

The variables used to define the method's three arguments were set at previous points in the application file:

- The **$config_format** variable was previously set to **xml** unless the **-t** command-line option was included:

  ```
  my $config_format = "xml";
  $config_format = "text" if $opt{t};
  ```

- The **$load_action** variable was previously set to **merge** unless the **-a** command-line option was included. The final two lines verify that the specified value is one of the acceptable choices:

  ```
  my $load_action = "merge";
  $load_action = $opt{a} if $opt{a};
  use constant VALID_ACTIONS => "merge|replace|override";
  output_usage() unless ( $load_action =~ /VALID_ACTIONS/);
  ```

- The **$doc** variable was set to the output from the **parsestring** or **parsefile** function (defined in the **XML::DOM::Parser** module), as described in "Reading In and Parsing the Configuration Data" on page 221.

The script performs two additional checks for errors and invokes the **graceful_shutdown** subroutine in either case:

```
unless ( ref $res ) {
  print "ERROR: Failed to load the configuration from $xmlfile\n";
  graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
$err = $res->getFirstError();
if ($err) {
  print "ERROR: Failed to load the configuration. Reason: $err->{message}\n";
  graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

Committing the Configuration

If there are no errors, the script invokes the **commit_configuration** method (defined in the file lib/JUNOS/Methods.pm in the Junos XML protocol Perl distribution):

```
print "Committing configuration from $xmlfile ...\n";
$res = $jnx->commit_configuration();
$err = $res->getFirstError();
```

```
if ($err) {
  print "ERROR: Failed to commit configuration. Reason: $err->{message}.\n";
  graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

Parsing and Formatting the Response from the Junos XML Protocol Server

As the last step in sending a request, the application verifies that there are no errors with the response from the Junos XML protocol server (see "Submitting a Request" on page 218). 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 text. If the response consists of configuration data, the application can store it as XML (the Junos XML tag elements generated by default from the Junos XML protocol server) or transform it into formatted ASCII text.

The following sections discuss parsing and formatting options:

Parsing and Formatting an Operational Response

The following code sample from the diagnose_bgp.pl and get_chassis_inventory.pl scripts uses XSLT to transform an operational response from the Junos XML protocol server into a more readable format. A detailed discussion of the functional subsections follows the complete code sample.

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

# Retrieve the XSLT file
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
  die "ERROR: XSLT file $xslfile does not exist";

#Get the xmlfile
my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);

my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");

if ($nm) {
  print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
  my $command = "xsltproc $nm $deviceinfo{hostname}.xml";

  $command .= "> $outputfile" if $outputfile;
  system($command);
  print "Done\n" if $outputfile;
  print "See $outputfile\n" if $outputfile;
}

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 = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
  die "ERROR: XSLT file $xslfile does not exist";
```

For examples of XSLT files, see the following directories in the Junos XML protocol Perl distribution:

- The examples/diagnose_bpg/xsl directory contains XSLT files for the diagnose_bpg.pl script: dhtml.xsl generates dynamic HTML, html.xsl generates HTML, and **text.xsl** generates ASCII text.
- The examples/get_chassis_inventory/xsl directory contains XSLT files for the get_chassis_inventory.pl script: chassis_inventory_csv.xsl generates a list of comma-separated values, chassis_inventory_html.xsl generates HTML, and chassis_inventory_xml.xsl generates XML.

The actual parsing operation begins by setting the variable **$xmlfile** to a filename of the form ***device-name*.xml** and invoking the **printToFile** function to write the Junos XML protocol server's response into the file (the **printToFile** function is defined in the **XML::DOM::Parser** module):

```
my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);
```

The next line invokes the **translateXSLtoRelease** function (defined in the **Junos::Response** module) to alter one of the namespace definitions in the XSLT file. 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 Junos XML operational response tag element includes a code representing the Junos OS version, such as **10.4R1** for the initial version of Junos OS Release 10.4. 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 **$xslfile** variable to match the value in the Junos XML protocol server's response. It assigns the resulting XSLT file to the **$nm** variable.

```
my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");
```

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

```
if ($nm) {
  print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
  my $command = "xsltproc $nm $deviceinfo{hostname}.xml";
```

If the **$outputfile** variable is defined (the file for storing the result of the XSLT transformation exists), the script appends a string to the **$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.

```
  $command .= "> $outputfile" if $outputfile;
  system($command);
  print "Done\n" if $outputfile;
  print "See $outputfile\n" if $outputfile;
}
```

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

```
else {
  print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

Parsing and Outputting Configuration Data

The get_config.pl script uses the **outconfig** subroutine to write the configuration data obtained from the Junos XML protocol server to a file either as Junos XML tag elements or as formatted ASCII text.

The **outconfig** subroutine takes four parameters. Three must have defined values: the directory in which to store the output file, device hostname, and the XML DOM tree (the configuration data) returned by the Junos XML protocol server. The fourth parameter indicates whether to output the configuration as formatted ASCII text, and has a null value if the requested output is Junos XML tag elements. In the following code sample, the script obtains values for the four parameters and passes them to the **outconfig** subroutine. A detailed discussion of each line follows the complete code sample.

```
my(%opt,$login,$password);

getopts('l:p:dm:hit', \%opt) || output_usage();
output_usage() if $opt{h};

my $basepath = shift || output_usage;

my $hostname = shift || output_usage;

my $config = getconfig( $hostname, $jnx, $opt{t} );
```

```
outconfig( $basepath, $hostname, $config, $opt{t} );
```

In the first lines of the preceding sample code, the get_config.pl script uses the following statements to obtain values for the four parameters to the **outconfig** subroutine:

- If the user provides the **-t** option on the command line, the **getopts** subroutine records it in the **%opt** hash. The value keyed to **$opt{t}** is passed as the fourth parameter to the **outconfig** subroutine. (For more information about reading options from the command line, see “Example: Collecting Parameters Interactively” on page 210.)

  ```
  getopts('l:p:dm:hit', \%opt) || output_usage();
  ```

- The following line reads the first element of the command line that is not an option preceded by a hyphen. It assigns the value to the **$basepath** variable, defining the name of the directory in which to store the file containing the output from the **outconfig** subroutine. The variable value is passed as the first parameter to the **outconfig** subroutine.

  ```
  my $basepath = shift || output_usage;
  ```

- The following line reads the next element on the command line. It assigns the value to the **$hostname** variable, defining the routing, switching, or security device hostname. The variable value is passed as the second parameter to the **outconfig** subroutine.

  ```
  my $hostname = shift || output_usage;
  ```

- The following line invokes the **getconfig** subroutine to obtain configuration data from the Junos XML protocol server on the specified device, assigning the resulting XML DOM tree to the **$config** variable. The variable value is passed as the third parameter to the **outconfig** subroutine.

  ```
  my $config = getconfig( $hostname, $jnx, $opt{t} );
  ```

The following code sample invokes and defines the **outconfig** subroutine. A detailed discussion of each functional subsection in the subroutine follows the complete code sample.

```
outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$$ ) {
  my $leader = shift;
  my $hostname = shift;
  my $config = shift;
  my $text_mode = shift;
  my $trailer = "xmlconfig";
  my $filename = $leader . "/" . $hostname . "." . $trailer;

  print "# storing configuration for $hostname as $filename\n";

  my $config_node;
  my $top_tag = "configuration";
  $top_tag .= "-text" if $text_mode;
  if ($config->getTagName() eq $top_tag) {
    $config_node = $config;
  } else {
    print "# unknown response component ", $config->getTagName(), "\n";
```

```
    }

    if ( $config_node && $config_node ne "" ) {
      if ( open OUTPUTFILE, ">$filename" )   {
        if (!$text_mode) {
          print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
          print OUTPUTFILE $config_node->toString(), "\n";
        } else {
          my $buf = $config_node->getFirstChild()->toString();
          $buf =~ s/($char_class)/$escapes{$1}/ge;
          print OUTPUTFILE "$buf\n";
        }
        close OUTPUTFILE;
      }
      else {
        print "ERROR: could not open output file $filename\n";
      }
    }
    else {
      print "ERROR: empty configuration data for $hostname\n";
    }
  }
```

The first lines of the **outconfig** subroutine read in the four parameters passed in when the subroutine is invoked, assigning each to a local variable:

```
outconfig( $basepath, $hostname, $config, $opt{t} );
sub outconfig( $$$$ ) {
  my $leader = shift;
  my $hostname = shift;
  my $config = shift;
  my $text_mode = shift;
```

The subroutine constructs the name of the file to which to write the subroutine's output and assigns the name to the **$filename** variable. The filename is constructed from the first two parameters (the directory name and hostname) and the **$trailer** variable, resulting in a name of the form ***directory-name/hostname*.xmlconfig**:

```
  my $trailer = "xmlconfig";
  my $filename = $leader . "/" . $hostname . "." . $trailer;

  print "# storing configuration for $hostname as $filename\n";
```

The subroutine checks that the first tag in the XML DOM tree correctly indicates the type of configuration data in the file. If the user included the **-t** option on the command line, the first tag should be **<configuration-text>** because the file contains formatted ASCII configuration statements; otherwise, the first tag should be **<configuration>** because the file contains Junos XML tag elements. The subroutine sets the **$top_tag** variable to the appropriate value depending on the value of the **$text_mode** variable (which takes its value from **opt{t}**, passed as the fourth parameter to the subroutine). The subroutine invokes the **getTagName** function (defined in the **XML::DOM::Element** module) to retrieve the name of the first tag in the input file, and compares the name to the value of the **$top_tag** variable. If the comparison succeeds, the XML DOM tree is assigned to the **$config_node** variable. Otherwise, the subroutine prints an error message because the XML DOM tree is not valid configuration data.


```
my $config_node;
my $top_tag = "configuration";
$top_tag .= "-text" if $text_mode;
if ($config->getTagName( ) eq $top_tag) {
  $config_node = $config;
} else {
  print "# unknown response component ", $config->getTagName( ), "\n";
}
```

The subroutine then uses several nested **if** statements. The first **if** statement verifies that the XML DOM tree exists and contains data:

```
if ( $config_node && $config_node ne "" ) {
  ... actions if XML DOM tree contains data ...
}
else {
  print "ERROR: empty configuration data for $hostname\n";
}
```

If the XML DOM tree contains data, the subroutine verifies that the output file can be opened for writing:

```
if ( open OUTPUTFILE, ">$filename" )   {
  ... actions if output file is writable ...
}
else {
  print "ERROR: could not open output file $filename\n";
}
```

If the output file can be opened for writing, the script writes the configuration data into it. If the user requested Junos XML tag elements—the user did not include the **-t** option on the command line, so the **$text_mode** variable does not have a value—the script writes the string **<?xml version=1.0?>** as the first line in the output file, and then invokes the **toString** function (defined in the **XML::DOM** module) to write each Junos XML tag element in the XML DOM tree on a line in the output file:

```
if (!$text_mode) {
  print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
  print OUTPUTFILE $config_node->toString( ), "\n";
```

If the user requested formatted ASCII text, the script invokes the **getFirstChild** and **toString** functions (defined in the **XML::DOM** module) to write the content of each tag on its own line in the output file. The script substitutes predefined entity references for disallowed characters (which are defined in the **%escapes** hash), writes the output to the output file, and closes the output file. (For information about defining the **%escapes** hash to contain the set of disallowed characters, see “Converting Disallowed Characters” on page 214.)

```
} else {
  my $buf = $config_node->getFirstChild( )->toString( );
  $buf =~ s/($char_class)/$escapes{$1}/ge;
  print OUTPUTFILE "$buf\n";
```



```
}
close OUTPUTFILE;
```

Closing the Connection to the Junos XML Protocol Server

To end the Junos XML protocol session and close the connection to the device, each sample script invokes the **request_end_session** and **disconnect** methods. Several of the scripts do this in standalone statements:

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

The load_configuration.pl script invokes the **graceful_shutdown** subroutine instead (for more information, see “Handling Error Conditions” on page 220):

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

Mapping CLI Commands to Perl Methods

The sample scripts described in “Overview of the Junos::Device Perl Module and Sample Scripts” on page 203 invoke only a small number of the predefined Junos XML Perl methods available in the current version of the Junos OS. There is a Perl method for every Junos XML request tag element. To derive the Perl method name from the request tag element name, replace each hyphen in the tag element name with an underscore and remove the enclosing angle brackets from the tag element name. For example, the **get_bgp_group_information** Perl method corresponds to the **<get-bgp-group-information>** tag element.

For a list of all of the Perl methods available in the current version of the Junos OS, see the chapter in the *Junos XML API Operational Reference* that maps Junos XML request tag elements to CLI commands and Perl methods. For information about optional and required attributes for a particular Perl method, see the entry for the corresponding Junos XML request tag element in the chapter titled “ Summary of Operational Request Tags” in the *Junos XML API Operational Reference*.

CHAPTER 10

Writing Junos XML Protocol C Client Applications

In this section, we offer two examples of using C to create client applications to access routers, switches, and security devices running Junos OS. The first example shows how to establish a Junos XML protocol session. The second example, shows how to retrieve and manipulate device configurations using C.

Establishing a Junos XML Protocol Session

The following example illustrates how a client application written in C can use the SSH or Telnet protocol to establish a Junos XML protocol connection and session. In the line that begins with the string **execlp**, the client application invokes the **ssh** command. (Substitute the **telnet** command if appropriate.) The ***routing-platform*** argument to the **execlp** routine specifies the hostname or IP address of the Junos XML protocol server device. The **junoscript** argument is the command that converts the connection to a Junos XML protocol session.

For more information about Junos XML protocol sessions, see "Controlling the Junos XML Management Protocol Session" on page 27.

```
int ipipes[ 2 ], opipes[ 2 ];
pid_t pid;
int rc;
char buf[ BUFSIZ ];

if (pipe(ipipes) <0 || pipe(opipes) <0)
  err(1, "pipe failed");

pid = fork( );
if (pid <0)
  err(1, "fork failed");

if (pid == 0) {
  dup2(opipes[ 0 ], STDIN_FILENO);
  dup2(ipipes[ 1 ], STDOUT_FILENO);
  dup2(ipipes[ 1 ], STDERR_FILENO);
```

```
    close(ipipes[ 0 ]); /* close read end of pipe */
    close(ipipes[ 1 ]); /* close write end of pipe */
    close(opipes[ 0 ]); /* close read end of pipe */
    close(opipes[ 1 ]); /* close write end of pipe */

    execlp("ssh", "ssh", "-x",  routing-platform  , "junoscript", NULL);
    err (1, "unable to execute: ssh %s junoscript," device);
  }

  close(ipipes[ 1 ]); /* close write end of pipe */
  close(opipes[ 0 ]); /* close read end of pipe */

  if (write(opipes[ 1 ], initial_handshake, strlen(initial_handshake)) <0 )
    err(1, "writing initial handshake failed");

  rc=read(ipipes[ 0 ], buf, sizeof(buf));
  if (rc <0)
     err(1, "read initial handshake failed");
```

Accessing and Editing Device Configurations

This example code shows a script that can be used to access, manipulate and commit device configurations using C.

```
//--Includes--//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>

//--Defines--//
//#define PRINT
   //--Toggles printing of all data to and from js server--//

//--Global Variables and Initialization--//
int sockfd;
char *xmlns_start_ptr = NULL;
char *xmlns_end_ptr = NULL;
int sock_bytes, pim_output_len, igmp_output_len, count_a, count_x, count_y,
      count_z, repl_str_len, orig_len, up_to_len, remain_len, conf_chg;
struct sockaddr_in serv_addr;
struct hostent *server;
char temp_buff[1024];        //--Temporary buffer used when --//
                             //--sending js configuration commands--//
char rcvbuffer[255];         //--Stores data from socket--//
char *pim_output_ptr = NULL;   //--Pointer for pim_output from socket--//
                             //--buffer--//
char *igmp_output_ptr = NULL;   //--Pointer for igmp_output from socket buffer--//
```

```
char small_buff[2048];       //--Buffer to support js communication--//
char jserver[16];            //--Junos XML protocol server IP address--//
int jport = 3221;            //--Junos XML protocol server port --//
                             //--(xnm-clear-text)--//
char msource[16];            //--Multicast source of group being
                             //--configured under igmp--//
char minterface[16];         //--Local multicast source interface--//
                             //--###change in igmp_xpath_ptr as well###--//
xmlDocPtr doc;               //--Pointer struct for parsing XML--//
xmlChar *pim_xpath_ptr =
      (xmlChar*) "/rpc-reply/pim-join-information/join-family
       /join-group[upstream-state-flags/local-source]
       /multicast-group-address";
xmlChar *temp_xpath_ptr =
      (xmlChar*) "/rpc-reply/igmp-group-information
      /mgm-interface-groups/mgm-group
      [../interface-name = '%s']/multicast-group-address";
xmlChar *igmp_xpath_ptr = NULL;
xmlNodeSetPtr nodeset;
xmlXPathObjectPtr pim_result;   //--Pointer for pim result xml parsing--//
xmlXPathObjectPtr igmp_result;  //--Pointer for igmp result xml parsing--//
xmlChar *keyword_ptr = NULL;    //--Pointer for node text--//
char pim_result_buff[128][64];  //--Char array to store pim xPath results--//
char igmp_result_buff[128][64]; //--Char array to store igmp xPath results--//

//--js commands-//
char js_handshake1[64] = "<?xml version=\"1.0\" encoding=\"us-ascii\"?>\n";
char js_handshake2[128] = "<junoscript version=\"1.0\"
      hostname=\"client1\" release=\"8.4R1\">\n";
char js_login[512] = "<rpc>\n<request-login>\n<username>lab</username>
      \n<challenge-response>Lablab</challenge-response>
      \n</request-login>\n</rpc>\n";
char js_show_pim[512] = "<rpc>\n<get-pim-join-information>
      \n<extensive/></get-pim-join-information></rpc>\n";
char js_show_igmp[512] = "<rpc>\n<get-igmp-group-information/>\n</rpc>\n";
char js_rmv_group[512] = "<rpc>\n<load-configuration>\n<configuration>
      \n<protocols>\n<igmp>\n<interface>\n<name>%s</name>
      \n<static>\n<group delete='delete'>\n<name>%s</name>
      \n</group>\n</static>\n</interface>\n</igmp>\n</protocols>
      \n</configuration>\n</load-configuration>\n</rpc>\n\n\n\n\n";
char js_add_group[512] = "<rpc>\n<load-configuration>
      \n<configuration>\n<protocols>\n<igmp>
      \n<interface>\n<name>%s</name>\n<static>
      \n<group>\n<name>%s</name>\n<source>
      \n<name>%s</name>\n</source>\n</group>\n</static>
      \n</interface>\n</igmp>\n</protocols>\n</configuration>
      \n</load-configuration>\n</rpc>\n";
char js_commit[64] = "<rpc>\n<commit-configuration/>\n</rpc>\n";

//--Function prototypes--//
void error(char *msg);       //--Support error messaging--//
xmlDocPtr getdoc(char *buffer); //--Parses XML content and loads it into memory--//
xmlXPathObjectPtr getnodeset (xmlDocPtr doc, xmlChar *xpath);
      //--Parses xml content for result node(s) from XPath search--//

//--Functions--//
```

```
void error(char *msg) {
 perror(msg);
 exit(0);
}

xmlDocPtr getdoc(char *buffer) {

 xmlDocPtr doc;

 doc = xmlReadMemory(buffer, strlen((char *)buffer), "temp.xml", NULL, 0);
  if (doc == NULL ) {
   fprintf(stderr,"Document not parsed successfully. \n");
   return NULL;
  } else {
   #ifdef PRINT
   printf("Document parsed successfully. \n");
   #endif
  }
  return doc;
 }

 xmlXPathObjectPtr getnodeset (xmlDocPtr doc, xmlChar *xpath) {

  xmlXPathContextPtr context;
  xmlXPathObjectPtr result;

  context = xmlXPathNewContext(doc);
  if (context == NULL) {
   printf("Error in xmlXPathNewContext\n");
   return NULL;
  }
  result = xmlXPathEvalExpression(xpath, context);
  xmlXPathFreeContext(context);
  if (result == NULL) {
   printf("Error in xmlXPathEvalExpression\n");
   return NULL;
  }
  if(xmlXPathNodeSetIsEmpty(result->nodesetval)) {
   xmlXPathFreeObject(result);
   #ifdef PRINT
   printf("No result\n");
   #endif
   return NULL;
  }
  return result;
 }

//--Main--//
 int main(int argc, char **argv) {

  if(argc != 4) {
   printf("\nUsage: %s <device Address> <Interface Name>
        <Multicast Source>\n\n", argv[0]);
   exit(0);
  } else {
   strcpy(jserver, argv[1]);
```

```
 strcpy(minterface, argv[2]);
 strcpy(msource, argv[3]);
}
igmp_xpath_ptr = (xmlChar *)realloc((xmlChar *)igmp_xpath_ptr, 1024);
sprintf(igmp_xpath_ptr, temp_xpath_ptr, minterface);

sockfd = socket(AF_INET, SOCK_STREAM, 0);
server = gethostbyname(jserver);
bzero((char*) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char*) server->h_addr, (char*)
      &serv_addr.sin_addr.sin_addr, server->h_length);
serv_addr.sin_port = htons(jport);

//--Connect to the js server--//
if(connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
 printf("Socket connect error\n");
}

if(fcntl(sockfd, F_SETOWN, getpid()) < 0)
error("Unable to set process owner to us\n");
printf("\nConnected to %s on port %d\n", jserver, jport);

//--Read data from the initial connect--//
sock_bytes = read(sockfd, rcvbuffer, 255);
#ifdef PRINT
printf("\n%s", rcvbuffer);
#endif

//--js intialization handshake--//
sock_bytes = write(sockfd, js_handshake1, strlen(js_handshake1));
      //--Send xml PI to js server--//
sock_bytes = write(sockfd, js_handshake2, strlen(js_handshake2));
      //--Send xml version and encoding to js server--//
sock_bytes = read(sockfd, rcvbuffer, 255);
      //--Read return data from sock buffer--//
rcvbuffer[sock_bytes] = 0;
printf("XML connection to the Junos XML protocol server has been initialized\n");
#ifdef PRINT
printf("\n%s", rcvbuffer);
#endif

//--js login--//
sock_bytes = write(sockfd, js_login, strlen(js_login));
      //--Send js command--//
while(strstr(small_buff, "superuser") == NULL) {
      //--Continue to read from the buffer until match--//
 sock_bytes = read(sockfd, rcvbuffer, 255);
 rcvbuffer[sock_bytes] = 0;
 strcat(small_buff, rcvbuffer);
       //--Copy buffer contents into pim_buffer--//
}
printf("Login completed to the Junos XML protocol server\n");
#ifdef PRINT
printf("%s\n", small_buff);  //--Print the small buff contents--//
#endif
```

```
//regfree(&regex_struct);
 bzero(small_buff, strlen(small_buff));
       //--Erase small buffer contents--//

//--Begin the for loop here--//
 printf("Running continuous IGMP and PIM group comparison...\n\n");
 for(;;) {              //--Begin infinite for loop--//

//--Get PIM join information--//
  pim_output_ptr = (char *)realloc((char *)pim_output_ptr,
        strlen(js_handshake1));
        //--Allocate memory for xml PI concatenation --//
        //--to pim_output_ptr--//
  strcpy(pim_output_ptr, js_handshake1);
        //--Copy PI to pim_output_ptr--//
  sock_bytes = write(sockfd, js_show_pim, strlen(js_show_pim));
        //--Send show pim joins command--//
  while(strstr(pim_output_ptr, "</rpc-reply>") == NULL) {
        //--Continue to read from the buffer until match--//
   sock_bytes = read(sockfd, rcvbuffer, 255);
         //--Read from buffer--//
   rcvbuffer[sock_bytes] = 0;
   pim_output_len = strlen((char *)pim_output_ptr);
         //--Determine current string length of pim_output_ptr--//
   pim_output_ptr = (char *)realloc((char *)pim_output_ptr,
         strlen(rcvbuffer)+pim_output_len);
         //--Reallocate memory for additional data--//
   strcat(pim_output_ptr, rcvbuffer);
         //--Copy data from rcvbuffer to pim_output_ptr--//
  }

//--Remove the xmlns entry--//
  xmlns_start_ptr = strstr(pim_output_ptr, "xmlns=\"http:");
        //--Find the start of the xmlns entry--pointer --//
        //--returned by strstr()--//
  xmlns_end_ptr = strstr(xmlns_start_ptr, ">");
        //--Find the end of the xmlns entry--pointer --//
        //--returned by strstr()--//
  repl_str_len = xmlns_end_ptr - xmlns_start_ptr;
        //--Determine the length of the string to be replaced--//
  orig_len = strlen((char *)pim_output_ptr) + 1;
        //--Determine the original length of pim_output--//
  up_to_len = xmlns_start_ptr - pim_output_ptr;
        //--Determine the length up to the beginning --//
        //--of the xmlns entry--//
  remain_len = orig_len - (up_to_len + repl_str_len);
        //--Determine what the remaining length is minus --//
        //--what we are removing--//
  memcpy(xmlns_start_ptr - 1, xmlns_start_ptr + repl_str_len, remain_len);
        //--copy the remaining string to the beginning --//
        //--of the replacement string--//
  #ifdef PRINT
  printf("\n%s\n", pim_output_ptr);
  #endif
//--End of GET PIM join information--//
```

```
//--Get IGMP membership information--//
  igmp_output_ptr = (char *)realloc((char *)igmp_output_ptr,
       strlen(js_handshake1));
  strcpy(igmp_output_ptr, js_handshake1);
  sock_bytes = write(sockfd, js_show_igmp, strlen(js_show_igmp));
  while(strstr(igmp_output_ptr, "</rpc-reply>") == NULL) {
   sock_bytes = read(sockfd, rcvbuffer, 255);
   rcvbuffer[sock_bytes] = 0;
   igmp_output_len = strlen((char *)igmp_output_ptr);
   igmp_output_ptr = (char *)realloc((char *)igmp_output_ptr,
        strlen(rcvbuffer)+igmp_output_len);
   strcat(igmp_output_ptr, rcvbuffer);
  }
  #ifdef PRINT
  printf("\n%s\n", igmp_output_ptr);
  #endif
//--End of GET IGMP membership information--//

//--Store xPath results for pim buffer search--//
  doc = getdoc(pim_output_ptr);
       //--Call getdoc() to parse XML in pim_output--//
  pim_result = getnodeset (doc, pim_xpath_ptr);
       //--Call getnodeset() which provides xPath result--//
  if (pim_result) {
   nodeset = pim_result->nodesetval;
   for (count_a=0; count_a < nodeset->nodeNr; count_a++) {
        //--Run through all node values found--//
    keyword_ptr = xmlNodeListGetString
         (doc, nodeset->nodeTab[count_a]->xmlChildrenNode, 1);
    strcpy(pim_result_buff[count_a], (char *)keyword_ptr);
         //--Copy each node value to its own array element--//
   #ifdef PRINT
   printf("PIM Groups: %s\n", pim_result_buff[count_a]);
        //--Print the node value--//
   #endif

   xmlFree(keyword_ptr);    //--Free memory used by keyword_ptr--//
   xmlChar *keyword_ptr = NULL;
   }
  xmlXPathFreeObject(pim_result);
       //--Free memory used by result--//
 }
 xmlFreeDoc(doc);          //--Free memory used by doc--//
 xmlCleanupParser();       //--Clean everything else--//
//--End of xPath search--//

//--Store xPath results for igmp buffer search--//
 doc = getdoc(igmp_output_ptr);
 igmp_result = getnodeset (doc, igmp_xpath_ptr);
 if (igmp_result) {
  nodeset = igmp_result->nodesetval;
  for (count_a=0; count_a < nodeset->nodeNr; count_a++) {
   keyword_ptr = xmlNodeListGetString
        (doc, nodeset->nodeTab[count_a]->xmlChildrenNode, 1);
   strcpy(igmp_result_buff[count_a], (char *)keyword_ptr);
   #ifdef PRINT
```

```
    printf("IGMP Groups: %s\n", igmp_result_buff[count_a]);
    #endif

    xmlFree(keyword_ptr);
    xmlChar *keyword_ptr = NULL;
   }
   xmlXPathFreeObject(igmp_result);
  }
  xmlFreeDoc(doc);
  xmlCleanupParser();
 //--End of xPath search--//

 //--Code to compare pim groups to configured igmp static membership--//
  conf_chg = 0;
  count_x=0;              //--Track pim groups--//
  count_y=0;              //--Track igmp groups--//
  count_z=0;       //--Track matches (if set to 1, igmp group matched pim group)--//
  while(strstr(pim_result_buff[count_x], "2") != NULL) {
        //--Run through igmp pim groups--//
   if(strstr(igmp_result_buff[count_y], "2") == NULL) {
    count_z = 0;
    conf_chg = 1;
   }
   while(strstr(igmp_result_buff[count_y], "2") != NULL) {
         //--For each pim group, run through all igmp groups--//
    if(strcmp(igmp_result_buff[count_y], pim_result_buff[count_x]) == 0) {
          //--If igmp group matches pim group, set z to 1 --//
          //-- (ie  count_z=1; --//
          //--Set z to 1 if there was a match (ie - the static --//
          //--membership is configured)--//
    }
    count_y++;          //--Increment igmp result buffer--//
   }
   if(count_z == 0) {      //--If no igmp group matched the --//
                    //--pim group (z stayed at 0), configure--//
                    //--static membership--//
    printf("Adding this group to igmp: %s\n", pim_result_buff[count_x]);
    sprintf(temp_buff, js_add_group, minterface,
         pim_result_buff[count_x], msource);
         //--Copy js_add_group with pim group to temp_buff--//
    #ifdef PRINT
    printf("%s", temp_buff);
    #endif
    sock_bytes = write(sockfd, temp_buff, strlen(temp_buff));
    while(strstr(small_buff, "</rpc-reply>") == NULL) {
     sock_bytes = read(sockfd, rcvbuffer, 255);
     rcvbuffer[sock_bytes] = 0;
     strcat(small_buff, rcvbuffer);
    }
    #ifdef PRINT
    printf("%s\n", small_buff);
    #endif
    bzero(small_buff, strlen(small_buff));
          //--Erase (copy all 0's) small buffer contents--//
    bzero(temp_buff, strlen(temp_buff));
          //--Erase temp_buff contents--//
```

```
    conf_chg = 1;
          //--Set conf_chg value to 1 to signify that a --//
          //--commit is needed--//
   }
   count_x++;          //--increment pim result buffer--//
   count_y=0;          //--reset igmp result buffer to start--//
                       //-- at first element--//
   count_z=0;          //--reset group match to 0 --//
                       //--(config needed due to no match)--/
  }

 //--Code for comparing igmp static membership to pim groups--//
  count_x=0;
  count_y=0;
  count_z=0;
  while(strstr(igmp_result_buff[count_y], "2") != NULL) {
   if(strstr(pim_result_buff[count_x], "2") == NULL) {
    count_z = 0;
    conf_chg = 1;
   }
   while(strstr(pim_result_buff[count_x], "2") != NULL) {
    if(strcmp(pim_result_buff[count_x], igmp_result_buff[count_y]) == 0) {
     count_z = 1;
    }
    count_x++;
   }
   if(count_z == 0) {
    printf("Removing this group from igmp: %s\n", igmp_result_buff[count_y]);
    sprintf(temp_buff, js_rmv_group, minterface, igmp_result_buff[count_y]);
    #ifdef PRINT
    printf("%s", temp_buff);
    #endif
    sock_bytes = write(sockfd, temp_buff, strlen(temp_buff));
    while(strstr(small_buff, "</rpc-reply>") == NULL) {
     sock_bytes = read(sockfd, rcvbuffer, 255);
     rcvbuffer[sock_bytes] = 0;
     strcat(small_buff, rcvbuffer);
    }
    #ifdef PRINT
    printf("%s\n", rcvbuffer);
    #endif
    bzero(small_buff, strlen(small_buff));
    bzero(temp_buff, strlen(temp_buff));
    conf_chg = 1;
   }
   count_y++;
   count_x=0;
   count_z=0;
  }

  if(conf_chg == 1) {
   sock_bytes = write(sockfd, js_commit, strlen(js_commit));
   while(strstr(small_buff, "</rpc-reply>") == NULL) {
    sock_bytes = read(sockfd, rcvbuffer, 255);
    rcvbuffer[sock_bytes] = 0;
    strcat(small_buff, rcvbuffer);
```

```
    }
    bzero(small_buff, strlen(small_buff));
    printf("\nCommitted configuration change\n");
   } else {
    #ifdef PRINT
    printf("\nNo configuration changes made\n");
    #endif
   }
   #ifdef PRINT
   printf("\n%s\n", small_buff);
   #endif

  //--Cleanup before next round of checks--//
   bzero(rcvbuffer, strlen(rcvbuffer));
         //--Erase contents of rcvbuffer--//
   char *xmlns_start_ptr = NULL;
         //--Nullify the contents--//
   char *xmlns_end_ptr = NULL;
         //--Nullify the contents--//
   for(count_x = 0; count_x < 129; count_x++) {
         //--Erase contents of both pim_result_buff and igmp_result_buff--//
    bzero(pim_result_buff[count_x], strlen(pim_result_buff[count_x]));
    bzero(igmp_result_buff[count_x], strlen(igmp_result_buff[count_x]));
   }
  }
}
```

PART 4

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Index of Statements and Commands

T

U

V

X