



R3.3 Beta Draft—Cisco Confidential Information



Cisco IOS XR XML API Guide

| Cisco IOS XR Software Release 3.3

Corporate Headquarters

Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
<http://www.cisco.com>
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

| Text Part Number: OL-8522-01



THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CCSP, CCVP, the Cisco Square Bridge logo, Follow Me Browsing, and StackWise are trademarks of Cisco Systems, Inc.; Changing the Way We Work, Live, Play, and Learn, and iQuick Study are service marks of Cisco Systems, Inc.; and Access Registrar, Aironet, ASIST, BPX, Catalyst, CCDA, CCDP, CCIE, CCIP, CCNA, CCNP, Cisco, the Cisco Certified Internetwork Expert logo, Cisco IOS, Cisco Press, Cisco Systems, Cisco Systems Capital, the Cisco Systems logo, Cisco Unity, Empowering the Internet Generation, Enterprise/Solver, EtherChannel, EtherFast, EtherSwitch, Fast Step, FormShare, GigaDrive, GigaStack, HomeLink, Internet Quotient, IOS, IP/TV, iQ Expertise, the iQ logo, iQ Net Readiness Scorecard, LightStream, Linksys, MeetingPlace, MGX, the Networkers logo, Networking Academy, Network Registrar, *Packet*, PIX, Post-Routing, Pre-Routing, ProConnect, RateMUX, ScriptShare, SlideCast, SMARTnet, StrataView Plus, TeleRouter, The Fastest Way to Increase Your Internet Quotient, and TransPath are registered trademarks of Cisco Systems, Inc. and/or its affiliates in the United States and certain other countries.

All other trademarks mentioned in this document or Website are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (0502R)

Cisco IOS XR XML API Guide

Copyright © 2005 Cisco Systems, Inc. All rights reserved.

Document Revision History	x
About This Document	x
Intended Audience	x
Organization of the Document	x
Related Documentation	xi
Document Conventions	xi
Obtaining Documentation	xii
Cisco.com	xii
Product Documentation DVD	xii
Ordering Documentation	xii
Documentation Feedback	xiii
Cisco Product Security Overview	xiii
Reporting Security Problems in Cisco Products	xiii
Obtaining Technical Assistance	xiv
Cisco Technical Support & Documentation Website	xiv
Submitting a Service Request	xv
Definitions of Service Request Severity	xv
Obtaining Additional Publications and Information	xv
Cisco XML API Overview	17
Introduction	17
Definition of Terms	18
Cisco Management XML Interface	18
Cisco XML API and Router System Features	19
Cisco XML API Tags	20
Basic XML Request Content	20
XML Declaration Tag	21
Operation Type Tags	23
XML Request Batching	24
Cisco XML Router Configuration and Management	27
Target Configuration Overview	27
Configuration Operations	28
Additional Configuration Options Using XML	28

R3.3 Beta Draft—Cisco Confidential Information

Locking the Running Configuration	29
Browsing the Target or Running Configuration	29
Browsing the Changed Configuration	31
Loading the Target Configuration	33
Setting the Target Configuration Explicitly	33
Saving the Target Configuration	35
Committing the Target Configuration	36
Unlocking the Running Configuration	40
Additional Router Configuration and Management Options Using XML	41
Getting Commit Changes	41
Loading Commit Changes	43
Clearing a Target Session	44
Rolling Back Configuration Changes to a Specified Commit Identifier	45
Rolling Back Configuration Changes to a Specified Number of Commits	46
Getting Rollback Changes	47
Loading Rollback Changes	48
Getting Configuration History	50
Getting Configuration Session Information	51
Replacing the Current Running Configuration	52
Cisco XML Operational Requests and Fault Management	55
Operational Get Requests	55
Action Requests	56
Cisco XML and Fault Management	57
Cisco XML and Native Data Operations	59
Native Data Operation Content	59
Request Type Tag and Namespaces	60
Object Hierarchy	60
Dependencies Between Configuration Items	63
Null Value Representations	64
Operation Triggering	64
Native Data Operation Examples	65
Cisco XML and Native Data Access Techniques	71
Available Set of Native Data Access Techniques	71
XML Request for All Configuration Data	72
XML Request for All Configuration Data per Component	72
XML Request for All Data Within a Container	73
XML Request for Specific Data Items	74

R3.3 Beta Draft—Cisco Confidential Information

XML Request with Combined Object Class Hierarchies	75
XML Request Using Wildcarding (Match Attribute)	78
XML Request for Specific Object Instances (Repeated Naming Information)	80
XML Request Using Operation Scope (Content Attribute)	82
Limiting the Number of Table Entries Returned (Count Attribute)	84
Custom Filtering (Filter Element)	85

Cisco XML and Encapsulated CLI Operations 89

XML CLI Command Tags	89
CLI Command Limitations	90

Cisco XML and Large Data Retrieval (Iterators) 91

Terminating an Iterator	94
-------------------------	----

Cisco XML Security 97

Authentication	97
Authorization	97
Retrieving Task Permissions	98
Task Privileges	99
Task Names	99
Authorization Failure	100

Cisco XML Schema Versioning 101

Major and Minor Version Numbers	101
Run-Time Use of Version Information	102
Placement of Version Information	103
Version Lag	103
Version Creep	104
Retrieving Version Information	105

Alarms 107

Alarm Registration	107
Alarm Deregistration	108
Alarm Notification	109

Error Reporting in Cisco XML Responses 111

Types of Reported Errors	111
Error Attributes	112
Transport Errors	112
XML Parse Errors	112
XML Schema Errors	113

R3.3 Beta Draft—Cisco Confidential Information

Operation Processing Errors 115

Error Codes and Messages 115

XML Transport and Event Notifications 117

Cisco XML Transport and CORBA IDL 117

IDL Interface 118

Authentication 119

CORBA NameServer 120

Client Access to the Root Naming Context 120

LR Name Server Tree 120

Event Notifications and Alarms 121

CORBA Notification Structure 121

CORBA XML Agent Initiative 122

CORBA XML Limitations 122

XML Agent Errors 123

TTY-Based Transports 123

Enabling the TTY XML Agent 123

Enabling a Session from a Client 124

Sending XML Requests and Receiving Responses 124

Errors That Result in No XML Response Being Produced 124

Ending a Session 124

Summary of Cisco XML API Configuration Tags 125

Cisco XML Schemas 127

XML Schema Retrieval 127

Common XML Schemas 128

Component XML Schemas 128

Schema File Organization 128

Schema File Upgrades 129

Cisco IOS XR Perl Scripting Toolkit 131

Cisco IOS XR Perl Scripting Toolkit Concepts 132

Security Implications for the Cisco IOS XR Perl Scripting Toolkit 132

Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit 132

Installing the Cisco IOS XR Perl Scripting Toolkit 133

Using the Cisco IOS XR Perl XML API in a Perl Script 134

Handling Types of Errors for the Cisco IOS XR Perl XML API 134

Starting a Management Session on a Router 134

R3.3 Beta Draft—Cisco Confidential Information

Closing a Management Session on a Router	136
Sending an XML Request to the Router	136
Using Response Objects	136
Using the Error Objects	137
Using the Configuration Services Methods	138
Using the Cisco IOS XR Perl Data Object Interface	140
Understanding the Perl Data Object Documentation	141
Generating the Perl Data Object Documentation	141
Creating Data Objects	142
Specifying the Schema Version to Use When Creating a Data Object	143
Using the Data Operation Methods on a Data Object	144
Using the Batching API	147
Displaying Data and Keys Returned by the Data Operation Methods	148
Specifying the Session to Use for the Data Operation Methods	149
Cisco IOS XR Perl Notification and Alarm API	149
Registering for Alarms	150
Deregistering an Existing Alarm Registration	150
Deregistering All Registration on a Particular Session	150
Receiving an Alarm on a Management Session	150
Using the Debug and Logging Facilities	151
Debug Facility Overview	151
Logging Facility Overview	152
Examples of Using the Cisco IOS XR Perl XML API	153
Configuration Examples	154
Operational Examples	161

R3.3 Beta Draft—Cisco Confidential Information

Preface

Beta Draft Cisco Confidential Information

This document describes the extensible markup language (XML) application programming interface (API) provided by the router to developers of external management applications. The XML interface provides a mechanism for the router configuration and monitoring using XML formatted request and response streams.

The XML schemas referenced in this guide are used by the management application developer to integrate client applications with the router programmable interface. The XML API is also available for use on any Cisco platform running Cisco IOS XR software.



Note

The XML API code is available for use on any Cisco platform that runs Cisco IOS XR software.

The preface contains the following sections:

- [Document Revision History, page x](#)
- [About This Document, page x](#)
- [Related Documentation, page xi](#)
- [Document Conventions, page xi](#)
- [Obtaining Documentation, page xii](#)
- [Documentation Feedback, page xiii](#)
- [Cisco Product Security Overview, page xiii](#)
- [Obtaining Technical Assistance, page xiv](#)
- [Obtaining Additional Publications and Information, page xv](#)

R3.3 Beta Draft—Cisco Confidential Information

Document Revision History

The Document Revision History table records technical changes to this document. [Table 1](#) shows the document revision number for the change, the date of the change, and a brief summary of the change. Note that not all Cisco documents use a Document Revision History table.

Table 1 Document Revision History

Revision	Date	Change Summary
OL-8522-01	December 2005	This is the initial release of this document.

About This Document

This document provides a comprehensive description of the XML interface to the router. The goal of this guide is to help customers write client applications to interact with the XML infrastructure on the router, and use the XML API to build custom end-user interfaces for configuration and information retrieval and display.

Intended Audience

This document's audience is expected to have background knowledge and information about XML, Common Object Request Broker Architecture (CORBA), and network management.

This document will be of particular use to those who want to build management client applications.

Organization of the Document

This document consists of the following chapters:

- [Chapter 1, “Cisco XML API Overview”](#)
- [Chapter 2, “Cisco XML Router Configuration and Management”](#)
- [Chapter 3, “Cisco XML Operational Requests and Fault Management”](#)
- [Chapter 4, “Cisco XML and Native Data Operations”](#)
- [Chapter 5, “Cisco XML and Native Data Access Techniques”](#)
- [Chapter 6, “Cisco XML and Encapsulated CLI Operations”](#)
- [Chapter 7, “Cisco XML and Large Data Retrieval \(Iterators\)”](#)
- [Chapter 8, “Cisco XML Security”](#)
- [Chapter 9, “Cisco XML Schema Versioning”](#)
- [Chapter 10, “Alarms”](#)
- [Chapter 11, “Error Reporting in Cisco XML Responses”](#)
- [Chapter 12, “XML Transport and Event Notifications”](#)
- [Chapter 13, “Summary of Cisco XML API Configuration Tags”](#)
- [Chapter 14, “Cisco XML Schemas”](#)

R3.3 Beta Draft—Cisco Confidential Information

- [Chapter 15, “Cisco IOS XR Perl Scripting Toolkit”](#)
- [Appendix A, “Sample BGP Configuration”](#)

Related Documentation

The following list of related documents are useful:

- *Cisco IOS XR Getting Started Guide*
- *Cisco Craft Works Interface Configuration Guide*
- *Release Notes for Cisco IOS XR Software, Release 3.2*
- Cisco IOS XR software configuration guides and command references

Document Conventions

This publication uses the following conventions:

- The symbol ^ represents the key labeled Ctrl. For example, the key combination ^z means “hold down the Ctrl key while you press the Z key.”

Command descriptions use these conventions:

- Examples that contain system prompts denote interactive sessions, indicating the commands that you should enter at the prompt. The system prompt indicates the current level of the EXEC command interpreter. For example, the prompt Router> indicates the user level, and the prompt Router# indicates the privileged level. Access to the privileged level usually requires a password.
- Commands and keywords are in **boldface** font.
- Arguments for which you supply values are in *italic* font.
- Elements in square brackets ([]) are optional.
 - Alternative but required keywords are grouped in braces ({ }) and separated by vertical bars (|).

Examples use these conventions:

- Terminal sessions and sample console screen displays are in *screen* font.
- Information you enter is in **boldface screen** font.
- Nonprinting characters, such as passwords, are in angle brackets (< >) in environments in which italic font is not available.
- Default responses to system prompts are in square brackets ([]).
- An exclamation point (!) at the beginning of a line indicates a comment line.



Note

Means *reader take note*. Notes contain helpful suggestions or references to material not covered in the publication.



Tip

Means *the following information will help you solve a problem*. The information in tips might not contain troubleshooting or task-specific actions, but tips do contain useful information.

R3.3 Beta Draft—Cisco Confidential Information

**Caution**

Means *reader be careful*. In this situation, you might do something that could result in equipment damage or loss of data.

Obtaining Documentation

Cisco documentation and additional literature are available on Cisco.com. Cisco also provides several ways to obtain technical assistance and other technical resources. These sections explain how to obtain technical information from Cisco Systems.

Cisco.com

You can access the most current Cisco documentation at this URL:

<http://www.cisco.com/techsupport>

You can access the Cisco website at this URL:

<http://www.cisco.com>

You can access international Cisco websites at this URL:

http://www.cisco.com/public/countries_languages.shtml

Product Documentation DVD

Cisco documentation and additional literature are available in the Product Documentation DVD package, which may have shipped with your product. The Product Documentation DVD is updated regularly and may be more current than printed documentation.

The Product Documentation DVD is a comprehensive library of technical product documentation on portable media. The DVD enables you to access multiple versions of hardware and software installation, configuration, and command guides for Cisco products and to view technical documentation in HTML. With the DVD, you have access to the same documentation that is found on the Cisco website without being connected to the Internet. Certain products also have .pdf versions of the documentation available.

The Product Documentation DVD is available as a single unit or as a subscription. Registered Cisco.com users (Cisco direct customers) can order a Product Documentation DVD (product number DOC-DOCDVD=) from Cisco Marketplace at this URL:

<http://www.cisco.com/go/marketplace/>

Ordering Documentation

Beginning June 30, 2005, registered Cisco.com users may order Cisco documentation at the Product Documentation Store in the Cisco Marketplace at this URL:

<http://www.cisco.com/go/marketplace/>

R3.3 Beta Draft—Cisco Confidential Information

Nonregistered Cisco.com users can order technical documentation from 8:00 a.m. to 5:00 p.m. (0800 to 1700) PDT by calling 1 866 463-3487 in the United States and Canada, or elsewhere by calling 011 408 519-5055. You can also order documentation by e-mail at tech-doc-store-mkpl@external.cisco.com or by fax at 1 408 519-5001 in the United States and Canada, or elsewhere at 011 408 519-5001.

Documentation Feedback

You can rate and provide feedback about Cisco technical documents by completing the online feedback form that appears with the technical documents on Cisco.com.

You can send comments about Cisco documentation to bug-doc@cisco.com.

You can submit comments by using the response card (if present) behind the front cover of your document or by writing to the following address:

Cisco Systems
Attn: Customer Document Ordering
170 West Tasman Drive
San Jose, CA 95134-9883

We appreciate your comments.

Cisco Product Security Overview

Cisco provides a free online Security Vulnerability Policy portal at this URL:

http://www.cisco.com/en/US/products/products_security_vulnerability_policy.html

From this site, you can perform these tasks:

- Report security vulnerabilities in Cisco products.
- Obtain assistance with security incidents that involve Cisco products.
- Register to receive security information from Cisco.

A current list of security advisories and notices for Cisco products is available at this URL:

<http://www.cisco.com/go/psirt>

If you prefer to see advisories and notices as they are updated in real time, you can access a Product Security Incident Response Team Really Simple Syndication (PSIRT RSS) feed from this URL:

http://www.cisco.com/en/US/products/products_psirt_rss_feed.html

Reporting Security Problems in Cisco Products

Cisco is committed to delivering secure products. We test our products internally before we release them, and we strive to correct all vulnerabilities quickly. If you think that you might have identified a vulnerability in a Cisco product, contact PSIRT:

- Emergencies—security-alert@cisco.com

An emergency is either a condition in which a system is under active attack or a condition for which a severe and urgent security vulnerability should be reported. All other conditions are considered nonemergencies.

R3.3 Beta Draft—Cisco Confidential Information

- Nonemergencies—psirt@cisco.com

In an emergency, you can also reach PSIRT by telephone:

- 1 877 228-7302
- 1 408 525-6532



Tip

We encourage you to use Pretty Good Privacy (PGP) or a compatible product to encrypt any sensitive information that you send to Cisco. PSIRT can work from encrypted information that is compatible with PGP versions 2.x through 8.x.

Never use a revoked or an expired encryption key. The correct public key to use in your correspondence with PSIRT is the one linked in the Contact Summary section of the Security Vulnerability Policy page at this URL:

http://www.cisco.com/en/US/products/products_security_vulnerability_policy.html

The link on this page has the current PGP key ID in use.

Obtaining Technical Assistance

Cisco Technical Support provides 24-hour-a-day award-winning technical assistance. The Cisco Technical Support & Documentation website on Cisco.com features extensive online support resources. In addition, if you have a valid Cisco service contract, Cisco Technical Assistance Center (TAC) engineers provide telephone support. If you do not have a valid Cisco service contract, contact your reseller.

Cisco Technical Support & Documentation Website

The Cisco Technical Support & Documentation website provides online documents and tools for troubleshooting and resolving technical issues with Cisco products and technologies. The website is available 24 hours a day, at this URL:

<http://www.cisco.com/techsupport>

Access to all tools on the Cisco Technical Support & Documentation website requires a Cisco.com user ID and password. If you have a valid service contract but do not have a user ID or password, you can register at this URL:

<http://tools.cisco.com/RPF/register/register.do>



Note

Use the Cisco Product Identification (CPI) tool to locate your product serial number before submitting a web or phone request for service. You can access the CPI tool from the Cisco Technical Support & Documentation website by clicking the **Tools & Resources** link under Documentation & Tools. Choose **Cisco Product Identification Tool** from the Alphabetical Index drop-down list, or click the **Cisco Product Identification Tool** link under Alerts & RMAs. The CPI tool offers three search options: by product ID or model name; by tree view; or for certain products, by copying and pasting **show** command output. Search results show an illustration of your product with the serial number label location highlighted. Locate the serial number label on your product and record the information before placing a service call.

R3.3 Beta Draft—Cisco Confidential Information

Submitting a Service Request

Using the online TAC Service Request Tool is the fastest way to open S3 and S4 service requests. (S3 and S4 service requests are those in which your network is minimally impaired or for which you require product information.) After you describe your situation, the TAC Service Request Tool provides recommended solutions. If your issue is not resolved using the recommended resources, your service request is assigned to a Cisco engineer. The TAC Service Request Tool is located at this URL:

<http://www.cisco.com/techsupport/servicerequest>

For S1 or S2 service requests or if you do not have Internet access, contact the Cisco TAC by telephone. (S1 or S2 service requests are those in which your production network is down or severely degraded.) Cisco engineers are assigned immediately to S1 and S2 service requests to help keep your business operations running smoothly.

To open a service request by telephone, use one of the following numbers:

Asia-Pacific: +61 2 8446 7411 (Australia: 1 800 805 227)

EMEA: +32 2 704 55 55

USA: 1 800 553-2447

For a complete list of Cisco TAC contacts, go to this URL:

<http://www.cisco.com/techsupport/contacts>

Definitions of Service Request Severity

To ensure that all service requests are reported in a standard format, Cisco has established severity definitions.

Severity 1 (S1)—Your network is “down,” or there is a critical impact to your business operations. You and Cisco will commit all necessary resources around the clock to resolve the situation.

Severity 2 (S2)—Operation of an existing network is severely degraded, or significant aspects of your business operation are negatively affected by inadequate performance of Cisco products. You and Cisco will commit full-time resources during normal business hours to resolve the situation.

Severity 3 (S3)—Operational performance of your network is impaired, but most business operations remain functional. You and Cisco will commit resources during normal business hours to restore service to satisfactory levels.

Severity 4 (S4)—You require information or assistance with Cisco product capabilities, installation, or configuration. There is little or no effect on your business operations.

Obtaining Additional Publications and Information

Information about Cisco products, technologies, and network solutions is available from various online and printed sources.

- Cisco Marketplace provides a variety of Cisco books, reference guides, documentation, and logo merchandise. Visit Cisco Marketplace, the company store, at this URL:

<http://www.cisco.com/go/marketplace/>

R3.3 Beta Draft—Cisco Confidential Information

- *Cisco Press* publishes a wide range of general networking, training and certification titles. Both new and experienced users will benefit from these publications. For current Cisco Press titles and other information, go to Cisco Press at this URL:

<http://www.ciscopress.com>

- *Packet* magazine is the Cisco Systems technical user magazine for maximizing Internet and networking investments. Each quarter, Packet delivers coverage of the latest industry trends, technology breakthroughs, and Cisco products and solutions, as well as network deployment and troubleshooting tips, configuration examples, customer case studies, certification and training information, and links to scores of in-depth online resources. You can access Packet magazine at this URL:

<http://www.cisco.com/packet>

- *iQ Magazine* is the quarterly publication from Cisco Systems designed to help growing companies learn how they can use technology to increase revenue, streamline their business, and expand services. The publication identifies the challenges facing these companies and the technologies to help solve them, using real-world case studies and business strategies to help readers make sound technology investment decisions. You can access iQ Magazine at this URL:

<http://www.cisco.com/go/iqmagazine>

or view the digital edition at this URL:

<http://ciscoiq.texterity.com/ciscoiq/sample/>

- *Internet Protocol Journal* is a quarterly journal published by Cisco Systems for engineering professionals involved in designing, developing, and operating public and private internets and intranets. You can access the Internet Protocol Journal at this URL:

<http://www.cisco.com/ipj>

- Networking products offered by Cisco Systems, as well as customer support services, can be obtained at this URL:

<http://www.cisco.com/en/US/products/index.html>

- Networking Professionals Connection is an interactive website for networking professionals to share questions, suggestions, and information about networking products and technologies with Cisco experts and other networking professionals. Join a discussion at this URL:

<http://www.cisco.com/discuss/networking>

- World-class networking training is available from Cisco. You can view current offerings at this URL:

<http://www.cisco.com/en/US/learning/index.html>

Cisco XML API Overview

Beta Draft Cisco Confidential Information

This chapter contains the following sections:

- [Introduction, page 1-17](#)
- [Cisco Management XML Interface, page 1-18](#)
- [Cisco XML API and Router System Features, page 1-19](#)
- [Cisco XML API Tags, page 1-20](#)

Introduction

The *Cisco IOS XR XML API Guide* explains how to use the Cisco XML API to configure routers or request information about configuration, management, or operation of the routers. The goal of this guide is to help router customers write client applications to interact with the Cisco XML infrastructure on the router, and to also use the Management XML API to build custom end-user interfaces for configuration and information retrieval and display.

The extensible markup language (XML) application programming interface (API) provided by the router is an interface used for rapid development of client applications and perl scripts to manage and monitor the router. The XML interface is specified by the XML schemas. The XML API provides a mechanism for router configuration and monitoring utilizing an exchange of XML formatted request and response streams.

Client applications can be used to configure the router, or to request status information from the router, by encoding a request in XML API tags and sending it to the router. The router processes the request and sends the response to the client by again encoding the response in XML API tags. This guide describes the XML requests that can be sent by external client applications to access router management data, and also details the responses to the client by the router.

The XML API readily supports available transport layers including the Common Object Request Broker Architecture (CORBA) and the terminal-based protocols Telnet and Secure Shell (SSH).

Customers use a variety of vendor-specific command line interface (CLI) scripts to manage their routers because no alternative programmatic mechanism is available. In addition, a common framework has not been available to develop CLI scripts. In response to this need, the XML API provides the necessary common framework for rapid development, deployment, and maintenance of router management.



Note

The XML API code is available for use on any Cisco platform that runs Cisco IOS XR software.

R3.3 Beta Draft—Cisco Confidential Information

Definition of Terms

Table 1-1 defines the words, acronyms, and actions used throughout this guide.

Table 1-1 **Definition of Terms**

Term	Description
AAA	Authentication, authorization, and accounting
CLI	Command-line interface
CORBA	Common Object Request Broker Architecture
CWI	Craft Works Interface
SSH	Secure Shell
XML	Extensible markup language
XML agent	A process on the router that receives XML requests by XML clients, and is responsible to carry out the actions contained in the request and to return an XML response to the client.
XML client	An external application that sends XML requests to the router and receives XML responses to those requests.
XML operation	A portion of an XML request that specifies an operation that the XML client wants the XML agent to perform.
XML operation provider	The code that carries out a particular XML operation including parsing the operation XML, performing the operation, and assembling the operation XML response.
XML request	An XML document sent to the router containing a number of requested operations to be carried out.
XML response	The response to an XML request.
XML schema	An XML document specifying the structure and possible contents of XML elements that can be contained in an XML document.

Cisco Management XML Interface

The following information is listed regarding the Cisco Management XML interface:

- High-level structure of the XML request and response streams
- Operation tag types and usage, including their XML format and content

R3.3 Beta Draft—Cisco Confidential Information

- How to use XML to configure the router:
 - Using the two-stage “target configuration” mechanism provided by the configuration manager
 - Using features such as locking, loading, browsing, modifying, saving, and committing the configuration
- Accessing the router’s operational data with XML
- Working with the native management data object class hierarchies:
 - To represent the native data objects in XML
 - To use various techniques for structuring XML requests to access the management data of interest, including the use of wildcards and filters
- Encapsulating CLI commands in XML
- How error information is returned to the client application
- Using iterators for large data retrieval
- Handling event notifications with XML
- How authorization of client requests is enforced
- Versioning of the XML schemas
- Generation and packaging of the XML schemas
- Transporting options enable the corresponding XML agents on the router
- How to use the Cisco IOS XR Perl Scripting Toolkit to write Perl scripts to manage a Cisco IOS XR router

Cisco XML API and Router System Features

Using the XML API, an external client application can send XML encoded management requests to an XML agent running on the router. The XML API readily supports available transport layers including the Common Object Request Broker Architecture (CORBA) and the terminal-based protocols Telnet and Secure Shell (SSH). The Cisco XML API interface described in this document applies equally to all supported transports.

Before an XML session is established, the XML transport and XML agent must be enabled on the router. For more information, see [Chapter 12, “XML Transport and Event Notifications.”](#)

A client request sent to the router must specify the different types of operations to be carried out. The following three general types of management operations are supported through XML:

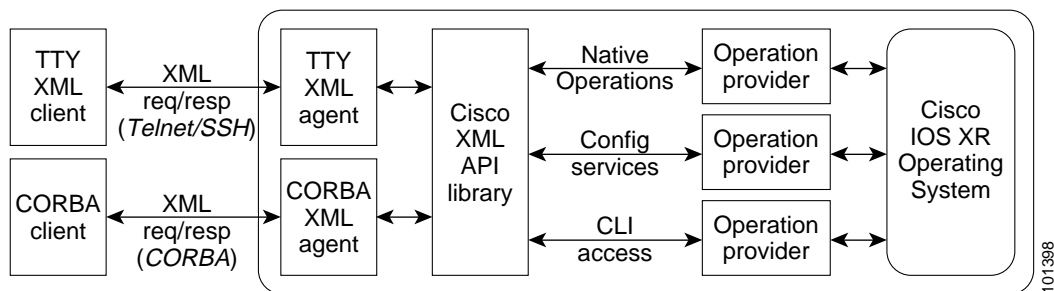
- Native data access (get, set, delete, and so on) using the native management data model.
- Configuration services for more advanced configuration management through the Configuration Manager.
- Traditional CLI access where the CLI commands and command responses are encapsulated in XML.

When a client request is received by an XML agent on the router, the request is routed to the appropriate XML operation provider in the internal Cisco XML API library for processing. After all the requested operations are processed, the XML agent receives the result and then sends the XML encoded response stream on to the client.

[Figure 1-1](#) presents a high level view of the XML manageability-related components along with the request and response flows between the client application and the router.

R3.3 Beta Draft—Cisco Confidential Information

Figure 1-1 XML Components and Request/Response Flows



Cisco XML API Tags

An external client application can access management data on the router through an exchange of well structured XML-tagged request and response streams. The XML tagged request and response streams are described in the following sections:

- [Basic XML Request Content, page 1-20](#)
- [XML Declaration Tag, page 1-21](#)
- [Operation Type Tags, page 1-23](#)
- [XML Request Batching, page 1-24](#)

Basic XML Request Content

This section describes the specific content and format of the XML data exchanged between the client and the router for the purpose of router configuration and monitoring.

Top-Level Structure

The top level of every request sent by a client application to the router must begin with an XML declaration tag followed by a request tag and one or more operation type tags. Similarly, every response returned by the router must begin with an XML declaration tag followed by a response tag and one or more operation type tags. Each request can contain operation tags for each supported operation type and the operation type tags can be repeated. The operation type tags contained in the response corresponds to those contained in the client request.

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Request from Client Application

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Operation>
    .
    .
    .
    Operation-specific content goes here
    .
    .
  </Operation>
</Request>
```

Sample XML Response from Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Operation>
    .
    .
    .
    Operation response data returned here
    .
    .
  </Operation>
</Response>
```



Note

All examples in this document are formatted with new lines and white space to aid readability. Actual XML request and response streams exchanged with the router do not include new lines and white space characters because these elements would add significantly to the size of the XML data and impact the overall performance of the XML API.

XML Declaration Tag

Each request and response exchanged between a client application and the router must begin with an XML declaration tag indicating which version of XML and (optionally) which character set is being used:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Table 1-2 defines the attributes of the XML declaration that are defined by the XML specification.

Table 1-2 Attributes for XML Declaration

Name	Description
Version	Specifies the version of XML to be used. Only Version “1.0” is supported by the router. Note The version attribute is required.
Encoding	Specifies the standardized character set to be used. Only “UTF-8” is supported by the router. The router includes the encoding attribute in a response only if it was specified in the corresponding request. Note The encoding attribute is optional.

R3.3 Beta Draft—Cisco Confidential Information

Request and Response Tags

Following the XML declaration tag a client application must enclose each request stream within a set of `<Request>` start and `</Request>` end tags. Also, the system encloses each XML response within a set of `<Response>` start and `</Response>` end tags. A major and minor version number are carried on the `<Request>` and `<Response>` elements to indicate the overall XML API version in use by the client application and router respectively.

The XML API presents a synchronous interface to client applications. The `<Request>` and `<Response>` tags are used by the client to correlate the request and response streams. A client application can issue a request after which the router returns a response. The client can then issue another request, and so on. Therefore, the XML session between a client and the router consist of a series of alternating requests and response streams.

The client application optionally includes a `ClientID` attribute within the `<Request>` tag. The value of the `ClientID` attribute must be an unsigned 32-bit integer value. If the `<Request>` tag contains a `ClientID` attribute, the router includes the same `ClientID` value in the corresponding `<Response>` tag. The `ClientID` value is treated as opaque data and is ignored by the router.

Maximum Request Size

The maximum size of an XML request or response is determined by the restrictions of the underlying transports. For more information on transport-specific limitations of request and response sizes, see [Chapter 12, “XML Transport and Event Notifications.”](#)

Minimum Response Content

If a `<Get>` request has nothing to return for a particular operation, the router returns the original request and an appropriate empty operation type tag. The minimum response returned by the router in the case of a single operation (for example, `<Get>`, `<Set>`, `<Delete>`) with no result data is shown in the following example.

Sample XML Request from Client Application

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Operation>
    .
    .
    .
    Operation-specific content goes here
    .
    .
    .
  </Operation>
</Request>
```

Sample XML Minimum Response from a Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Operation/>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

Operation Type Tags

Following the <Request> tag, the client application must specify the operation types to be carried out by the router. Three general types of operations are supported along with the <GetNext> operation for large responses.

Native Data Operation Tags

Native data operations provide basic access to the native management data model. [Table 1-3](#) describes the native data operation tags.

Table 1-3 Native Data Operation Tags

Native Data Tag	Description
<Get>	Get the value of one or more configuration, operational, or action data items.
<Set>	Create or modify one or more configuration or action data items.
<Delete>	Delete one or more configuration data items.
<GetVersionInfo>	Get the major and minor version numbers for one or more components.

The XML schema definitions for the native data operation type tags are contained in the schema file `native_data_operations.xsd`. The native data operations are described further in [Chapter 5, “Cisco XML and Native Data Access Techniques.”](#)

Configuration Services Operation Tags

Configuration services operations provide more advanced configuration management functions through the Configuration Manager. [Table 1-4](#) describes the configuration services operation tags.

Table 1-4 Configuration Services Operation Tags

Tag	Description
<Lock>	Lock the running configuration.
<Unlock>	Unlock the running configuration.
<Load>	Load the target configuration from a binary file previously saved by way of the <Save> tag.
<Save>	Save the target configuration to a binary file.
<Commit>	Promote the target configuration to the running configuration.
<Clear>	Abort/clear the current target configuration session.
<Rollback>	Roll back the running configuration to a previous configuration state.
<GetConfigurationHistory>	Get a list of commits made to the running configuration.
<GetConfigurationSessions>	Get a list of the user sessions currently configuring the box.

R3.3 Beta Draft—Cisco Confidential Information

The XML schema definitions for the configuration services operation type tags are contained in the schema file `config_services_operations.xsd` (see [Chapter 14, “Cisco XML Schemas”](#)).

The configuration services operations are described further in [Chapter 2, “Cisco XML Router Configuration and Management.”](#)

CLI Operation Tag

CLI access provides support for XML encapsulated CLI commands and responses. For CLI access, a single tag is provided. The `<CLI>` operation tag issues the request as a CLI command.

The XML schema definitions for the CLI tag are contained in the schema file `cli_operations.xsd` (see [Chapter 14, “Cisco XML Schemas”](#)).

The CLI operations are described further in [Chapter 6, “Cisco XML and Encapsulated CLI Operations.”](#)

GetNext Operation Tag

The `<GetNext>` tag is used to retrieve the next portion of a large response. It can be used as required to retrieve an oversize response following a request using one of the other operation types. The `<GetNext>` operation tag gets the next portion of a response. Iterators are supported for large requests.

The XML schema definition for the `<GetNext>` operation type tag is contained in the schema file `xml_api_protocol.xsd` (see [Chapter 14, “Cisco XML Schemas”](#)). For more information about the get next operation, see [Chapter 7, “Cisco XML and Large Data Retrieval \(Iterators\).”](#)

Alarm Operation Tags

The `<Alarm>` operation tag registers, unregisters, and receives alarm notifications. [Table 1-5](#) lists the alarm operation subtags.

Table 1-5 List of Alarm Operation Subtags

Subtag	Description
<code><Register></code>	Registers to receive alarm notifications.
<code><Unregister></code>	Cancels a previous alarm notification registration.

The XML schema definitions for the alarm operation tags are contained in the schema file `alarm_operations.xsd` (see [Chapter 14, “Cisco XML Schemas”](#)).

For more information about the alarm operations, see “Cisco XML Transport and CORBA IDL” section on page 12-117 of [Chapter 12, “XML Transport and Event Notifications.”](#)

XML Request Batching

The XML interface supports the combining of several requests or operations into a single request. When multiple operations are specified in a single request, the response contains the same operation tags and in the same order as they appeared in the request.

Batched requests are performed as a “best effort.” For example, if operations 1 through 3 are in the request and operation 2 fails, operation 3 is attempted.

R3.3 Beta Draft—Cisco Confidential Information

If you want to perform two or more <Get> operations, and the first one can return a large amount of data that is potentially larger than the size of one iterator chunk, you must place the subsequent operations within a separate XML request. If the operations are placed in the same request within the same <Get> tags, for example, potentially sharing part of the hierarchies with the first request, an error attribute that informs you that the operations cannot be serviced is returned on the relevant tags.

For more information, see [“XML Request with Combined Object Class Hierarchies”](#) section on page 5-75 of Chapter 5, “Cisco XML and Native Data Access Techniques.”

The following example shows a simple request containing six different operations:

Sample XML Client Batched Requests

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Lock/>
  <Get>
    <Configuration>
      .
      .
      .
      Get operation content goes here
      .
      .
      .
    </Configuration>
  </Get>
  <Set>
    <Configuration>
      .
      .
      .
      Set operation content goes here
      .
      .
      .
    </Configuration>
  </Set>
  <Commit/>
  <Get>
    <Operational>
      .
      .
      .
      Get operation content goes here
      .
      .
      .
    </Operational>
  </Get>
  <Unlock/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Lock/>
  <Get>
    <Configuration>
      .
      .
      .
      .
    </Configuration>
```

R3.3 Beta Draft—Cisco Confidential Information

```
.
.
.
.
.
Get response content returned here
.
.
.
.
.
.
.
</Configuration>
</Get>
<Set/>
<Commit CommitID="1000000188"/>
<Get>
    <Operational>
        .
        .
        .
        .
        .
        .
        .
        Get response content returned here
        .
        .
        .
        .
        .
        .
        .
    </Operational>
</Get>
<Unlock/>
</Response>
```

Cisco XML Router Configuration and Management

Beta Draft Cisco Confidential Information

This chapter reviews the basic extensible markup language (XML) requests and responses used to configure and manage the router.

The use of XML to configure the router is essentially an abstraction of a configuration editor in which client applications can, load, browse, and modify configuration data without affecting the current running (that is, active) configuration on the router. This configuration is called the "target configuration" and is not the running configuration on the router. The router's running configuration can never be modified directly. All changes to the running configuration must go through the target configuration.



Note

Each client application session has its own target configuration, which is not visible to other client sessions.

This chapter contains the following sections:

- [Target Configuration Overview, page 2-27](#)
- [Configuration Operations, page 2-28](#)
- [Additional Router Configuration and Management Options Using XML, page 2-41](#)

Target Configuration Overview

The target configuration is effectively the current running configuration overlaid with the client-entered configuration. In other words, the target configuration is the client-intended configuration if the client were to commit changes. In terms of implementation, the target configuration is an operating system buffer that contains just the changes (set and delete) that are performed within the configuration session.

A "client session" is synonymous with a single Common Object Request Broker Architecture (CORBA), Telnet, or Secure Shell (SSH) connection and authentication, authorization, and accounting (AAA) login. The target configuration is created implicitly at the beginning of a client application session and must be promoted (that is, committed) to the running configuration explicitly by the client application in order to replace or become the running configuration. If the client session breaks, the current target configuration is aborted and any outstanding locks are released.

R3.3 Beta Draft—Cisco Confidential Information**Note**

Only the syntax of the target configuration is checked and verified to be compatible with the installed software image on the router. The semantics of the target configuration is checked only when the target configuration is promoted to the running configuration.

Configuration Operations

**Note**

Only the tasks in the “[Committing the Target Configuration](#)” section are required to change the configuration on the router (that is, modifying and committing the target configuration).

Use the following configuration options from the client application to configure or modify the router with XML:

- [Locking the Running Configuration](#), page 2-29
- [Browsing the Target or Running Configuration](#), page 2-29
 - [Getting Configuration Data](#), page 2-30
- [Browsing the Changed Configuration](#), page 2-31
- [Loading the Target Configuration](#), page 2-33
- [Setting the Target Configuration Explicitly](#), page 2-33
- [Saving the Target Configuration](#), page 2-35
- [Committing the Target Configuration](#), page 2-36
 - [Loading a Failed Configuration](#), page 2-39
- [Unlocking the Running Configuration](#), page 2-40

Additional Configuration Options Using XML

Several optional configuration tasks are available to the client application to configure or modify the router with XML:

- [Getting Commit Changes](#), page 2-41
- [Clearing a Target Session](#), page 2-44
- [Rolling Back Configuration Changes to a Specified Commit Identifier](#), page 2-45
- [Rolling Back Configuration Changes to a Specified Number of Commits](#), page 2-46
- [Getting Rollback Changes](#), page 2-47
- [Getting Configuration History](#), page 2-50
- [Getting Configuration Session Information](#), page 2-51
- [Replacing the Current Running Configuration](#), page 2-52

R3.3 Beta Draft—Cisco Confidential Information

Locking the Running Configuration

The client application uses the <Lock> operation to obtain an exclusive lock on the running configuration in order to prevent modification by other users or applications.

If the lock operation is successful, the response contains only the <Lock/> tag. If the lock operation fails, the response also contains ErrorCode and ErrorMsg attributes that indicate the cause of the lock failure.

The following example shows a request to lock the running configuration. This request corresponds to the command-line interface (CLI) command **configure exclusive**.

Sample XML Request from the Client Application

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Lock/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Lock/>
</Response>
```

The following issues are used to lock the running configuration:

- The scope of the lock is the entire configuration “namespace.”
- Only one client application can hold the lock on the running configuration at a time. If a client application attempts to lock the configuration while another application holds the lock, an error is returned.
- If a client application has locked the running configuration, all other client applications can read only the running configuration, but cannot modify it (that is, they cannot commit changes to it).
- No mechanism is provided to allow a client application to break the lock of another user.
- If a client session is terminated, any outstanding locks are automatically released.
- The XML API does not support timeouts for locks.
- The <GetConfigurationSessions> operation is used to identify the user session holding the lock.

Browsing the Target or Running Configuration

The client application browses the target or current running configuration using the <Get> operation along with the <Configuration> request type tag. The client application optionally uses CLI commands encoded within XML tags to browse the configuration.

The <Configuration> tag supports the optional Source attribute, which is used to specify the source of the configuration information returned from a <Get> operation.

R3.3 Beta Draft – Cisco Confidential Information

Getting Configuration Data

Table 2-1 describes the Source options.

Table 2-1 Source Options

Name	Description
ChangedConfig	Read only from the changes made to the target configuration for the current session. This option effectively gets the configuration changes made from the current session since the last configuration commit. This option corresponds to the CLI command show configuration .
CurrentConfig	Read from the current active running configuration. This option corresponds to the CLI command show configuration running .
MergedConfig	Read from the target configuration for this session. This option should provide a view of the resultant running configuration if the current target configuration is committed without errors. For example, in the case of the “best effort” commit, some portions of the commit could fail, while others succeed. MergedConfig is the default when the Source attribute is not specified on the <Get> operation. This option corresponds to the CLI command show configuration merge .

If the get operation fails, the response contains one or more ErrorCode and ErrorMsg attributes indicating the cause of the failure.

The content and format of <Get> requests are described in additional detail in [Chapter 4, “Cisco XML and Native Data Operations.”](#) Encoding CLI commands within XML tags is described in [Chapter 6, “Cisco XML and Encapsulated CLI Operations.”](#)

The following example shows a <Get> request to browse the current Border Gateway Protocol (BGP) configuration:

Sample XML Client Request to Browse the Current BGP Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CurrentConfig">
      <BGP MajorVersion="1" MinorVersion="0"/>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CurrentConfig">
      <BGP MajorVersion="18" MinorVersion="0">
        ..
        .
        .
        response data goes here
        .
        .
        .
      </BGP>
    </Configuration>
  </Get>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

```
</Configuration>
</Get>
</Response>
```

Browsing the Changed Configuration

When a client application issues a <Get> request with a Source type of ChangedConfig, the response contains the OperationType attribute to indicate whether the returned changes to the target configuration were a result of <Set> or <Delete> operations.

Use <Get> to browse uncommitted target configuration changes.

The following example shows <Set> and <Delete> operations that modify the BGP configuration followed by a <Get> request to browse the uncommitted BGP configuration changes. These requests correspond to the following CLI commands:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router bgp 3
RP/0/RP0/CPU0:router(config-bgp)# default-metric 10
RP/0/RP0/CPU0:router(config-bgp)# no neighbor 10.0.101.8
RP/0/RP0/CPU0:router(config-bgp)# exit
RP/0/RP0/CPU0:router# show configuration
```

Sample XML to Modify the BGP Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
  <Delete>
    <Configuration>
      <BGP>
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>
                      <IPv4Address>10.0.101.8</IPv4Address>
                    </IPAddress>
                  </Naming>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Delete>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

```

        </DefaultVRF>
      </AS>
    </BGP>
  </Configuration>
</Delete>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
  <Delete>
    <Configuration/>
  </Delete>
</Response>

```

Sample XML Client Request to Browse Uncommitted Target Configuration Changes

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="ChangedConfig">
      <BGP MajorVersion="18" MinorVersion="0"/>
    </Configuration>
  </Get>
</Request>

```

Sample Secondary XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration OperationType="Set">
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
    <Configuration OperationType="Delete">
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>
                      <IPv4Address>10.0.101.8</IPv4Address>
                    </IPAddress>
                  </Naming>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>

```


R3.3 Beta Draft—Cisco Confidential Information

```

        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </DefaultVRF>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

Loading the Target Configuration

The client application uses the <Load> operation along with the <File> tag to populate the target configuration with the contents of a binary configuration file previously saved on the router using the <Save> operation.



Note

At the current time, a config file saved using CLI is not loadable with XML <Load>. The config should have been saved using the XML <Save> operation. Using the <Load> operation is strictly optional. It can be used alone or in conjunction with the <Set> and <Delete> operations as described in the section “Setting the Target Configuration Explicitly” on page 33

Use the <File> tag to name the file from which the configuration is to be loaded. When you use the <File> tag to name the file from which the configuration is to be loaded, specify the complete path of the file to be loaded.

If the load operation is successful, the response contains both the <Load> and <File> tags. If the load operation fails, the response can also contain the ErrorCode and ErrorMessage attributes that indicates the cause of the load failure.

The following example shows a request to load the target configuration from the contents of the file my_bgp.cfg:

Sample XML Client Request to Load the Target Configuration from a Named File

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <File>disk0:/my_bgp.cfg</File>
  </Load>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <File>disk0:/my_bgp.cfg</File>
  </Load>
</Response>

```

See also the “Setting the Target Configuration Explicitly” section on page 33.

Setting the Target Configuration Explicitly

The client application modifies the target configuration as needed using the <Delete> and <Set> operations.

R3.3 Beta Draft—Cisco Confidential Information


Note

There are not separate “Create” and “Modify” operations, because a <Set> operation for an item can result in the creation of the item if it does not already exist in the configuration, and a modification of the item if it does already exist.

The client application can optionally use CLI commands encoded within XML tags to modify the target configuration.

If the operation to modify the target configuration is successful, the response contains only the <Delete/> or <Set/> tag. If the operation fails, the response includes the element or object hierarchy passed in the request along with one or more ErrorCode and ErrorMsg attributes indicating the cause of the failure.

A syntax check is performed whenever the client application writes to the target configuration. A successful write to the target configuration, however, does not guarantee that the configuration change can succeed when a subsequent commit of the target configuration is attempted. For example, errors resulting from failed verifications may be returned from the commit. For information about the error returned from the XML API, see [Chapter 11, “Error Reporting in Cisco XML Responses.”](#)

The content and format of <Delete> and <Set> requests are described in additional detail in [Chapter 4, “Cisco XML and Native Data Operations.”](#)

Encoding CLI commands within XML tags is described in [Chapter 6, “Cisco XML and Encapsulated CLI Operations.”](#)

The following example shows how to use a <Set> request to set the default metric and routing timers and disable neighbor change logging for a particular BGP autonomous system. This request corresponds to the following CLI commands:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router bgp 3
RP/0/RP0/CPU0:router(config-bgp)# default-metric 10
RP/0/RP0/CPU0:router(config-bgp)# timers bgp 60 180
RP/0/RP0/CPU0:router(config-bgp)# bgp log neighbor changes
RP/0/RP0/CPU0:router(config-bgp)# exit
```

Sample XML Client Request to Set Timers and Disable Neighbor Change Logging for a BGP Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
              <GlobalTimers>
                <Keepalive>60</Keepalive>
                <Holdtime>180</Holdtime>
              </GlobalTimers>
              <DisableNbrLogging>true</DisableNbrLogging>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
</Response>
```

To replace a portion of the configuration, the client application should use a <Delete> operation to remove the unwanted configuration followed by a <Set> operation to add the new configuration. An explicit “replace” option is not supported.

For more information on replacing the configuration, see [“Replacing the Current Running Configuration” section on page 2-52](#).

Saving the Target Configuration

The client application uses the <Save> operation along with the <File> tag to save the contents of the target configuration to a binary file on the router.

Use the <File> tag to name the file to which the configuration is to be saved. You must specify the complete path of the file to be saved when you use the <File> tag. If the file already exists on the router, then an error is returned unless the optional Boolean attribute Overwrite is included on the <File> tag with a value of “true”.



Note

No mechanism is provided by the XML interface for “browsing” through the file directory structure.

If the save operation is successful, the response contains both the <Save> and <File> tags. If the save operation fails, the response can also contain the ErrorCode and ErrorMessage attributes that indicate the cause of the save failure.

The following example shows a request to save the contents of the target configuration to the file named my_bgp.cfg on the router:

Sample XML Client Request to Save the Target Configuration to a File

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Save>
    <File Overwrite="true">disk0:/my_bgp.cfg</File>
  </Save>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Save>
    <File Overwrite="true">disk0:/my_bgp.cfg</File>
  </Save>
</Response>
```

R3.3 Beta Draft – Cisco Confidential Information

Committing the Target Configuration

In order for the configuration in the target area to become part of the running configuration, the target configuration must be explicitly committed by the client application using the <Commit> operation.

Commit Operation


Table 2-2 describes the six optional attributes that are specified with the <Commit> operation.

Table 2-2 Commit Operation Attributes

Name	Description
Mode	Use the Mode attribute in the request to specify whether the target configuration should be committed on an Atomic or BestEffort basis. In the case of a commit with the Atomic option, the entire configuration in the target area is committed only if application of all the configuration in the target area to the running configuration succeeds. If any errors occur, the commit operation is rolled back and the errors are returned to the client application. In the case of commit with the BestEffort option, the configuration is committed even if some configuration items fail during the commit operation. In this case, the errors are also returned to the client application. By default, the commit operation is performed on an Atomic basis.
KeepFailedConfig	Use this boolean attribute to specify whether any configuration that fails during the commit operation should remain in the target configuration buffer. The default value for KeepFailedConfig is false. That is, by default the target configuration buffer is cleared after each commit. If a commit operation is performed with a KeepFailedConfig value of false, the user can then use the <Load> operation to load the failed configuration back into the target configuration buffer. The use of the KeepFailedConfig attribute makes sense only for the BestEffort commit mode. In the case of an Atomic commit, if something fails, the entire target configuration is kept intact (because nothing was committed).
Label	Use the Label attribute instead of the commit identifier wherever a commit identifier is expected, such as in the <Rollback> operation. The Label attribute is a unique user-specified label that is associated with the commit in the commit database. If specified, the label must begin with an alphabetic character and cannot match any existing label in the commit database.
Comment	Use the Comment attribute as a user-specified comment to be associated with the commit in the router commit database.

R3.3 Beta Draft—Cisco Confidential Information

Table 2-2 Commit Operation Attributes (continued)

Name	Description
Replace	<p>Use this boolean attribute to specify whether or not the commit operation should replace the entire router running configuration with the contents of the target configuration buffer. The default value for Replace is false. The Replace attribute should be used with caution.</p> <hr/> <p> Caution Note that the entire running configuration will be replaced by the contents of the target configuration buffer when a commit is issued with a Replace value of true. If the target configuration buffer does not contain the necessary configuration to maintain the management XML session with the router, the session will be terminated. This target configuration should include the appropriate interface and XML agent configuration.</p> <hr/>
IgnoreOtherSessions	<p>Use this boolean attribute to specify whether or not the commit operation should be allowed to go through without an error in the case where one or more commits have occurred from other configuration sessions since the current session started or since the last commit was made from this session. The default value for IgnoreOtherSessions is false.</p>

If the commit operation is successful, the response contains only the <Commit/> tag along with a unique CommitID and any other attributes specified in the request. If the commit operation fails, the failed configuration is returned in the response.

The following example shows a request to commit the target configuration using the Atomic option. The request corresponds to the CLI command **commit label BGPUpdate1 comment BGP config update**.

Sample XML Client Request to Commit the Target Configuration Using the Atomic Option

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Commit Mode="Atomic" Label="BGPUpdate1" Comment="BGP config update"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Commit Mode="Atomic" Label="BGPUpdate1"
    Comment="BGP config update"
    CommitID="1000000075"/>
</Response>
```

The following points should be noted with regard to committing the target configuration:

- After each successful commit operation, a commit record is created in the router commit database. The router maintains up to 100 entries in the commit database corresponding to the last 100 commits. Each commit is assigned a unique identifier, for example, "1000000075," which is saved with the commit information in the database. The commit identifier is used in subsequent operations such as <Get> commit changes or <Rollback> to a previous commit identifier (along with the <CommitID> tag).

R3.3 Beta Draft—Cisco Confidential Information

- The configuration changes in the target configuration are merged with the running configuration when committed. If a client application is to perform a replace of the configuration, the client must first remove the unwanted configuration using a <Delete> operation and then add the new configuration using a <Set> operation. An explicit replace option is not supported. For more information on replacing the configuration, see [“Replacing the Current Running Configuration” section on page 2-52](#).
- Applying the configuration for a trial period (“try-and-apply”) is not supported for this release.
- If the client application never commits, the target configuration is automatically destroyed when the client session is terminated. No other timeouts are supported.

Commit Errors

If any configuration entered into the target configuration fails to make its way to the running configuration as the result of a <Commit> operation (for example, the configuration contains a semantic error and is therefore rejected by a back-end application’s verifier function), then all of the failed configuration is returned in the <Commit> response along with the appropriate ErrorCode and ErrorMessage attributes indicating the cause of each failure.

The OperationType attribute is used to indicate whether the failure was a result of a requested <Set> or <Delete> operation. In the case of a <Set> operation failure, the value to be set is included in the commit response.

The following example shows <Set> and <Delete> operations to modify the BGP configuration followed by a <Commit> request resulting in failures for both requested operations. This request corresponds to the following CLI commands:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router bgp 4
RP/0/RP0/CPU0:router(config-bgp)# default-metric 10
RP/0/RP0/CPU0:router(config-bgp)# exit
RP/0/RP0/CPU0:router(config)# commit best-effort
```

Sample XML Client Request to Modify the Target Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>4</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
```

R3.3 Beta Draft—Cisco Confidential Information

```
<Configuration/>
</Set>
</Response>
```

Sample Request to Commit the Target Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Commit Mode="BestEffort"/>
</Request>
```

Sample XML Response from the Router Showing Failures for Both Requested Operations

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Commit Mode="BestEffort" ErrorCode="0x40819c00"
    ErrorMessage="&apos;sysdb&apos; detected the &apos;warning&apos; condition &apos;One
or more sub-operations failed during a best effort complex operation&apos; ">
    <Configuration OperationType="Set">
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>4</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric ErrorCode="0x409f8c00" ErrorMessage="AS number is wrong -
BGP is already running with AS number 3">10</DefaultMetric>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Commit>
</Response>
```

For more information, see [“Loading a Failed Configuration” section on page 2-39](#).

Loading a Failed Configuration

The client application uses the <Load> operation along with the <FailedConfig> tag to populate the target configuration with the failed configuration from the most recent <Commit> operation. Loading the failed configuration in this way is equivalent to specifying a “true” value for the KeepFailedConfig attribute in the <Commit> operation.

If the load is successful, the response contains both the <Load> and <FailedConfig> tags. If the load fails, the response can also contain the ErrorCode and ErrorMessage attributes that indicate the cause of the load failure.

The following example shows a request to load and display the failed configuration from the last <Commit> operation. This request corresponds to the CLI command **show configuration failed**.

Sample XML Client Request to Load the Failed Configuration from the Last <Commit> Operation

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <FailedConfig/>
  </Load>
  <Get>
    <Configuration Source="ChangedConfig"/>
  </Get>
```

R3.3 Beta Draft—Cisco Confidential Information

```
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <FailedConfig/>
  </Load>
  <Get>
    <Configuration OperationType="Set">
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>4</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Unlocking the Running Configuration

The client application must use the <Unlock> operation to release the exclusive lock on the running configuration for the current session prior to terminating the session.

If the unlock operation is successful, the response contains only the <Unock/> tag. If the unlock operation fails, the response can also contain the ErrorCode and ErrorMsg attributes that indicate the cause of the unlock failure.

The following example shows a request to unlock the running configuration. This request corresponds to the CLI command **exit** when it is used after the configuration mode is entered through the CLI command **configure exclusive**.

Sample XML Client Request to Unlock the Running Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Unlock/>
</Request>
```


R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Unlock/>
</Response>
```

Additional Router Configuration and Management Options Using XML

The following sections describe the optional configuration and router management tasks available to the client application:

- [Getting Commit Changes, page 2-41](#)
- [Loading Commit Changes, page 2-43](#)
- [Clearing a Target Session, page 2-44](#)
- [Rolling Back Configuration Changes to a Specified Commit Identifier, page 2-45](#)
- [Rolling Back Configuration Changes to a Specified Number of Commits, page 2-46](#)
- [Getting Rollback Changes, page 2-47](#)
- [Loading Rollback Changes, page 2-48](#)
- [Getting Configuration History, page 2-50](#)
- [Getting Configuration Session Information, page 2-51](#)
- [Replacing the Current Running Configuration, page 2-52](#)

Getting Commit Changes

When a client application successfully commits the target configuration to the running configuration, the configuration manager writes a single configuration change event to the system message logging (syslog). As a result, an event notification is written to the Alarm Channel (that is, the CORBA event notification channel for alarms) and subsequently forwarded to any registered configuration agents.

[Table 2-3](#) describes the event notification.

Table 2-3 Event Notification

Name	Description
userid	The name of the user who performed the commit operation.
timestamp	The date and time of the commit.
commit	The unique ID associated with the commit.

The following example shows a configuration change notification:

```
RP/0/RP0/CPU0:Jun 18 19:16:42.561 : %CLIENTLIBCFGMR-6-CONFIG_CHANGE : A configuration
commit by user 'root' occurred at 'Wed Jun 18 19:16:18 2004 '. The configuration changes
are saved on the router by commit ID: '1000000075'.
```

Upon receiving the configuration change notification, a client application can then use the <Get> operation to load and browse the changed configuration.

For more information on CORBA-based event notifications and alarms supported by the XML API, see [Chapter 12, “XML Transport and Event Notifications.”](#)

The following example shows the use of the `ForCommitID` attribute to show the commit changes for a specific commit. This request corresponds to the CLI command **show commit changes 1000000075**.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" SinceCommitID="1000000072">
      OperationType="....>
      .
      .
      changed config returned here
      .
      .
      .
```

R3.3 Beta Draft—Cisco Confidential Information

```
</Configuration>
</Get>
</Response>
```

Loading Commit Changes

The client application can load a set of commit changes into the target configuration buffer using the Load operation and CommitChanges tag along with one of the additional tags ForCommitID, SinceCommitID, or Previous. After the completion of the Load operation, the client application can then modify and commit the commit changes like any other config.

If the load is successful, the response contains both the Load and CommitChanges tags. If the load fails, the response also contains the ErrorCode and ErrorMessage attributes indicating the cause of the load failure.

The following example shows the use of the Load operation and CommitChanges tag along with the ForCommitID tag to load the commit changes for a specific commit into the target configuration buffer. This request corresponds to the CLI command **load commit changes 1000000072**.

Sample XML Request to Load Commit Changes with ForCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <ForCommitID>1000000072</ForCommitID>
    </CommitChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <ForCommitID>
        1000000072
      </ForCommitID>
    </CommitChanges>
  </Load>
</Response>
```

The following example shows the use of the Load operation and CommitChanges tag along with the SinceCommitID tag to load the commit changes since (and including) a specific commit into the target configuration buffer. This request corresponds to the CLI command **load commit changes since 1000000072**.

Sample XML Request to Load Commit Changes with SinceCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <SinceCommitID>1000000072</SinceCommitID>
    </CommitChanges>
  </Load>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <ForCommitID>
        1000000072
      </ForCommitID>
    </CommitChanges>
  </Load>
</Response>
```

The following example shows the use of the Load operation and CommitChanges tag along with the Previous tag to load the commit changes for the most recent four commits into the target configuration buffer. This request corresponds to the CLI command **load commit changes last 4**.

Sample XML Request to Load Commit Changes with Previous tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <Previous>4</Previous>
    </CommitChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges/>
  </Load>
</Response>
```

Clearing a Target Session

Prior to committing the target configuration to the active running configuration, the client application can use the <Clear> operation to clear the target configuration session. This operation has the effect of clearing the contents of the target configuration, thus removing any changes made to the target configuration since the last commit. The clear operation does not end the target configuration session, but results in the discarding of any uncommitted changes from the target configuration.

If the clear operation is successful, the response contains just the <Clear/> tag. If the clear operation fails, the response can also contain the ErrorCode and ErrorMessage attributes that indicate the cause of the clear failure.

The following example shows a request to clear the current target configuration session. This request corresponds to the CLI command **clear**.

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Request to Clear the Current Target Configuration Session

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Clear/>
</Request>
```

Sample XML Response from a Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Clear/>
</Response>
```

Rolling Back Configuration Changes to a Specified Commit Identifier

The client application uses the <Rollback> operation with the <CommitID> tag to roll back the configuration changes made since (and including) the commit by specifying a commit identifier or commit label.

If the roll back operation is successful, the response contains both the <Rollback> and <CommitID> tags. If the roll back operation fails, the response can also contain the ErrorCode and ErrorMsg attributes that indicate the cause of the roll back failure.

Table 2-4 describes the optional attributes that are specified with the <Rollback> operation by the client application when rolling back to a commit identifier.

Table 2-4 Optional Attributes for Rollback Operation (Commit Identifier)

Name	Description
Label	A unique user-specified label to be associated with the rollback in the router commit database. If specified, the label must begin with an alphabetic character and cannot match any existing label in the router commit database.
Comment	A user-specified comment to be associated with the rollback in the router commit database.

The following example shows a request to roll back the configuration changes to a specified commit identifier. This request corresponds to the CLI command **rollback configuration to 1000000072**.

Sample XML Request to Roll Back the Configuration Changes to a Specified Commit Identifier

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Rollback Label="BGPRollback1" Comment="My BGP rollback">
    <CommitID>1000000072</CommitID>
  </Rollback>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Rollback Label="BGPRollback1" Comment="My BGP rollback">
    <CommitID>1000000072</CommitID>
  </Rollback>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information


Note

The commit identifier can also be obtained by using the <GetConfigurationHistory> operation described in the section [“Getting Configuration History” section on page 2-50](#).

Rolling Back Configuration Changes to a Specified Number of Commits

The client application uses the <Rollback> operation with the <Previous> tag to roll back the configuration changes made during the most recent [x] commits, where [x] is a number ranging from 0 to the number of saved commits in the commit database. If the <Previous> value is specified as “0”, nothing is rolled back. The target configuration must be unlocked at the time the <Rollback> operation is requested.

If the roll back operation is successful, the response contains both the <Rollback> and <Previous> tags. If the roll back operation fails, the response can also contain the ErrorCode and ErrorMsg attributes that indicate the cause of the rollback failure.

[Table 2-5](#) describes the optional attributes that are specified with the <Rollback> operation by the client application when rolling back a specified number of commits.

Table 2-5 **Optional Attributes for Rollback Operation (Number of Commits)**

Name	Description
Label	A unique user-specified label to be associated with the rollback in the router commit database. If specified, the label must begin with an alphabetic character and cannot match any existing label in the router commit database.
Comment	A user-specified comment to be associated with the rollback in the router commit database.

The example shows a request to roll back the configuration changes made during the previous three commits. This request corresponds to the CLI command **rollback configuration last 3**.

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Request to Roll Back Configuration Changes to a Specified Number of Commits

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Rollback>
    <Previous>3</Previous>
  </Rollback>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Rollback>
    <Previous>3</Previous>
  </Rollback>
</Response>
```

Getting Rollback Changes

The client application can read a set of rollback changes using the <Get> operation along with the <Configuration> request type tag when it includes both the Source attribute option RollbackChanges and one of the additional attributes ToCommitID or PreviousCommits.

The set of roll back changes are the changes that are applied when the <Rollback> operation is performed using the same parameters. It is recommended that the client application read or verify the set of roll back changes before performing the roll back.

The following example shows the use of the ToCommitID attribute to get the rollback changes for rolling back to a specific commit. This request corresponds to the CLI command **show rollback-changes to 1000000072**.

Sample XML Client Request to Get Rollback Changes Using the ToCommitID Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="RollbackChanges" ToCommitID="1000000072"/>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="RollbackChanges" ToCommitID="1000000072">
      OperationType="...."
      .
      .
      rollback changes returned here
      .
      .
    </Configuration>
  </Get>
</Response>
```

The following example shows the use of the PreviousCommits attribute to get the roll back changes for rolling back a specified number of commits. This request corresponds to the CLI command **show rollback-changes last 4**.

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Client Request to Get Roll Back Changes Using the PreviousCommits Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="RollbackChanges" PreviousCommits="4"/>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="RollbackChanges" PreviousCommits="4">
      OperationType="..."
      .
      .
      rollback changes returned here
      .
      .
      .
    </Configuration>
  </Get>
</Response>
```

Loading Rollback Changes

The client application can load a set of rollback changes into the target configuration buffer using the Load operation and RollbackChanges tag along with one of the additional tags ForCommitID, ToCommitID, or Previous. After the completion of the Load operation, the client application can then modify and commit the rollback changes like any other configuration.

If the load is successful, the response contains both the Load and RollbackChanges tags. If the load fails, the response also contains the ErrorCode and ErrorMessage attributes indicating the cause of the load failure.

The following example shows the use of the Load operation and RollbackChanges tag along with the ForCommitID tag to load the rollback changes for a specific commit into the target configuration buffer. This request corresponds to the CLI command **load rollback changes 1000000072**.

Sample XML Client to Load Rollback Changes with ForCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges>
      <ForCommitID>1000000072</ForCommitID>
    </RollbackChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges/>
```


R3.3 Beta Draft—Cisco Confidential Information

```
</Load>
</Response>
```

The following example shows the use of the Load operation and RollbackChanges tag along with the ToCommitID tag to load the rollback changes up to (and including) a specific commit into the target configuration buffer. This request corresponds to the CLI command **load rollback changes to 1000000072**.

Sample XML Client to Load Rollback Changes with ToCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges>
      <ToCommitID>1000000072</ToCommitID>
    </RollbackChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges/>
  </Load>
</Response>
```

The following example shows the use of the Load operation and RollbackChanges tag along with the Previous tag to load the rollback changes for the most recent four commits into the target configuration buffer. This request corresponds to the CLI command **load rollback changes last 4**.

Sample XML Client to Load Rollback Changes with Previous tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges>
      <Previous>4</Previous>
    </RollbackChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges/>
  </Load>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

Getting Configuration History

The client application uses the `<GetConfigurationHistory>` operation to get information regarding the most recent commits to the running configuration.

Table 2-6 describes the information that is returned for each commit.

Table 2-6 Returned Commit Information

Returned Commit Information	Commit Information Description
<code><CommitID></code>	The unique ID associated with the commit.
<code><Label></code>	The optional label associated with the commit.
<code><UserID></code>	The name of the user who created the configuration session within which the commit was performed.
<code><Line></code>	The line used to connect to the router for the configuration session.
<code><ClientName></code>	The name of the client application that performed the commit.
<code><Timestamp></code>	The date and time of the commit.
<code><Comment></code>	The comment associated with the commit.

Table 2-7 describes the optional attributes available with the `<GetConfigurationHistory>` operation.

Table 2-7 Optional Attributes to Get Configuration History

Name	Description
Maximum	Use the Maximum attribute to specify the maximum number of entries to return from the commit history file. If the Maximum attribute is not included in the request or if the Maximum value is greater than the actual number of entries in the commit history file, all entries are returned. The commit entries are returned with the most recent commit first in the list.
RollbackOnly	Use the RollbackOnly Boolean attribute to specify whether the response should contain only those commits that can be rolled back. In addition to the commit history file, the router maintains a commit database of up to 100 records corresponding to the last 100 commits that can be rolled back. The default value for RollbackOnly is “false”. The <code><GetConfigurationHistory></code> operation used with a RollbackOnly value of “false” corresponds to the CLI command show configuration history . When the RollbackOnly attribute is specified as “true”, the operation corresponds to the CLI command show rollback-points .

The following example shows a request to list the information associated with the previous three commits. This request corresponds to the CLI command **show configuration commit history 3 detail**.

Sample XML Request to List Configuration History Information for the Previous Three Commits

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetConfigurationHistory Maximum="3"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
```

R3.3 Beta Draft—Cisco Confidential Information

```

<Response MajorVersion="1" MinorVersion="0">
  <GetConfigurationHistory Maximum="3">
    <CommitEntry>
      <Naming>
        <CommitID>1000000075</CommitID>
      </Naming>
      <Label>BGPUpdate1</Label>
      <UserID>cisco</UserID>
      <Line>line0</Line>
      <ClientName>XMLDemo</ClientName>
      <Timestamp>
        19:16:18 UTC Wed June 18 2003
      </Timestamp>
      <Comment>BGP config update</Comment>
    </CommitEntry>
    <CommitEntry>
      <Naming>
        <CommitID>1000000074</CommitID>
      </Naming>
      <Label xsi:nil="true">
      </Label>
      <UserID>unknown</UserID>
      <Line>con0_0_0</Line>
      <ClientName>CLI</ClientName>
      <Timestamp>
        11:07:55 UTC Tue June 17 2003
      </Timestamp>
      <Timestamp>Wed June 18 03:08:07 2003</Timestamp>
      <Comment xsi:nil="true">
      </Comment>
    </CommitEntry>
    <CommitEntry>
      <CommitID>1000000073</CommitID>
      <Label>MyCDPUpdate</Label>
      <UserID>nchomsky</UserID>
      <Line>line1</Line>
      <ClientName>XMLDemo</ClientName>
      <Timestamp>Tue June 17 11:07:55 2003</Timestamp>
      <Comment>My CDP config update</Comment>
    </CommitEntry>
  </GetConfigurationHistory>
</Response>

```

Getting Configuration Session Information

The client application uses the <GetConfigurationSessions> operation to get the list of all users configuring the router. In the case where the configuration is locked, the list identifies the user holding the lock.

Table 2-8 describes the information that is returned for each configuration session.

Table 2-8 Returned Session Information

Returned Session Information	Session Information Description
<SessionID>	The unique autogenerated ID for the configuration session.
<UserID>	The name of the user who created the configuration session.
<Line>	The line used to connect to the router.
<ClientName>	The user-friendly name of the client application that created the configuration session.

R3.3 Beta Draft—Cisco Confidential Information

Table 2-8 *Returned Session Information*

Returned Session Information	Session Information Description
<Since>	The date and time of the creation of the configuration session.
<LockHeld>	A Boolean operation indicating whether the session has an exclusive lock on the running configuration.

The following example shows a request to get the list of users configuring the router. This request corresponds to the CLI command **show configuration sessions**.

Sample XML Request to Get List of Users Configuring the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetConfigurationSessions/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetConfigurationSessions>
    <Session>
      <SessionID>
        00000070-001610ae-00000000
      </SessionID>
      <UserID>cisco</UserID>
      <Line>line0</Line>
      <ClientName>XMLDemo</ClientName>
      <Since>Fri Jun 27 15:10:39 2003</Since>
      <LockHeld>true</LockHeld>
    </Session>
    <Session>
      <Naming>
        <SessionID>
          00000070-001650a4-00000000
        </SessionID>
      </Naming>
      <UserID>unknown</UserID>
      <Line>con0_0_0</Line>
      <ClientName>XML-Agent</ClientName>
      <Since>Fri Jun 27 14:55:18 2003</Since>
      <LockHeld>false</LockHeld>
    </Session>
  </GetConfigurationSessions>
</Response>
```

Replacing the Current Running Configuration

A client application replaces the current running configuration on the router with an off-the-box configuration file by performing the following operations in sequence:

1. Lock the configuration.
2. Delete the entire configuration using a <Delete> operation along with the <Configuration/> tag.

R3.3 Beta Draft—Cisco Confidential Information

3. Load the desired off-the-box configuration into the target configuration using one or more <Set> operations (assuming that the entire desired configuration is available in XML format, perhaps from a previous <Get> of the entire configuration). As an alternative, use an appropriate **copy** command enclosed within <CLI> tags.
4. Commit the target configuration.

The following example illustrates these steps:

Sample XML Request to Lock the Current Running Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Lock/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Lock/>
</Response>
```

Sample XML Request to Delete the Current Running Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Delete>
    <Configuration/>
  </Delete>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Delete>
    <Configuration/>
  </Delete>
</Response>
```

Sample XML Request to Set the Current Running Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      .
      .
      .
      configuration data goes here
      .
      .
      .
    </Configuration>
  </Set>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
```

R3.3 Beta Draft – Cisco Confidential Information

```
</Response>
```

Sample XML Request to Commit the Target Configuration

```
<xml...>  
<Request...>  
  <Commit/>  
</Request>
```

Sample XML Response from the Router

```
<xml...>  
<Response...>  
  <Commit CommitID="1000000075"/>  
</Response>
```

Cisco XML Operational Requests and Fault Management

Beta Draft Cisco Confidential Information

A client application can send an extensible markup language (XML) request to get the router operational information using either a native data <Get> request along with the <Operational> tag, or the equivalent command-line interface (CLI) command. Although the CLI is more familiar to users, the advantage of using the <Get> request is that the response data is encoded in XML format instead of being only uninterpreted text enclosed within <CLI> tags.

This chapter contains the following sections:

- [Operational Get Requests, page 3-55](#)
- [Action Requests, page 3-56](#)

Operational Get Requests

The content and format of operational <Get> requests are described in additional detail in [Chapter 4, “Cisco XML and Native Data Operations.”](#)

The following example shows a <Get> request to retrieve the global Border Gateway Protocol (BGP) process information. This request returns BGP process information similar to that displayed by the **show ip bgp process detail** CLI command.

Sample XML Client Request to Get BGP Information

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Operational>
      <BGP MajorVersion="15" MinorVersion="0">
        <ProcessInfoTable/>
      </BGP>
    </Operational>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Operational>
```

R3.3 Beta Draft—Cisco Confidential Information

```

<BGP MajorVersion="15" MinorVersion="0">
  <ProcessInfoTable>
    <ProcessInfo>
      <Naming>
        <ProcessID>0</ProcessID>
      </Naming>
      <ProcessInstance>
        0
      </ProcessInstance>
      <ProcessInstanceNode>
        node0_0_CPU0
      </ProcessInstanceNode>
      <NeighborsCount>
        10
      </NeighborsCount>
      <EstablishedNeighborsCount>
        0
      </EstablishedNeighborsCount>
      <RestartCount>
        1
      </RestartCount>
      ....
      more response content her
      ...
    </ProcessInfo>
  </ProcessInfoTable>
</BGP>
</Operational>
</Get>
</Response>

```

Action Requests

A client application can send a <Set> request along with the <Action> tag to trigger unique actions on the router. For example, an object may be set with an action request to inform the router to clear a particular counter or reset some functionality. Most often this operation involves setting the value of a Boolean object to “true”. The content and format of <Set> requests are described in additional detail in [Chapter 4, “Cisco XML and Native Data Operations.”](#)

The following example shows an action request to clear the BGP performance statistics information. This request is equivalent to the **clear ip bgp performance-statistics** CLI command.

Sample XML Request to Clear BGP Performance Statistics Information

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Action>
      <BGP MajorVersion="3" MinorVersion="0">
        <ClearPerformanceStats>true</ClearPerformanceStats>
      </BGP>
    </Action>
  </Set>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Action/>
  </Set>
</Response>

```


R3.3 Beta Draft—Cisco Confidential Information

```
</Set>
</Response>
```

In addition, the following example is showing an action request to clear the peer drop information for all BGP neighbors. This request is equivalent to the CLI command **clear ip bgp peer-drops ***.

Sample XML Request to Clear Peer Drop Information for All BGP Neighbors

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Action>
      <BGP MajorVersion="3" MinorVersion="0">
        <ClearDrops>
          <All>true</All>
        </ClearDrops>
      </BGP>
    </Action>
  </Set>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Action/>
  </Set>
</Response>
```

Cisco XML and Fault Management

When a client application successfully commits the target configuration to the router's running configuration, the configuration manager writes a single configuration change event to system message logging (syslog). As a result, a fault management event notification is written to the Alarm Channel (that is, the Common Object Request Broker Architecture [CORBA] event notification channel for alarms) and subsequently forwarded to any registered configuration agents.

Configuration Change Notification

Table 3-1 provides event notification for configuration changes information.

Table 3-1 Event Notifications for Configuration Changes

Event Notification	Description
userid	The name of the user who performed the commit operation.
timestamp	The date and time of the commit.
commit	The unique ID associated with the commit.

The following example shows a configuration change notification:

```
RP/0/RP0/CPU0:Sep 18 09:43:42.747 : %CLIENTLIBCFGMGR-6-CONFIG_CHANGE : A configuration
commit by user root occurred at 'Wed Sep 18 09:43:42 2004 '. The configuration changes are
saved on the router in file: 010208180943.0
```

R3.3 Beta Draft – Cisco Confidential Information

Upon receiving the configuration change notification, a client application can then use the <Load> and <Get> operations to load and browse the changed configuration.

Cisco XML and Native Data Operations

Beta Draft Cisco Confidential Information

Native data operations <Get>, <Set>, and <Delete> provide basic access to configuration and operational data residing on the router.

This chapter describes the content of the native data operations and provides an example of each operation type.

Native Data Operation Content

The content of native data operations includes the request type and relevant object class hierarchy as described in the following sections.

The operations are described the following sections:

- [Request Type Tag and Namespaces, page 4-60](#)
- [Object Hierarchy, page 4-60](#)
- [Dependencies Between Configuration Items, page 4-63](#)
- [Null Value Representations, page 4-64](#)
- [Operation Triggering, page 4-64](#)
- [Native Data Operation Examples, page 4-65](#)

The following example shows a native data operation request:

Sample XML Client Native Data Operation Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Operation>
    <Request Type>
      .
      .
      .
      object hierarchy goes here
      .
      .
      .
    </Request Type>
  </Operation>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Operation>
    <Request Type>
      .
      .
      .
      response content returned here
      .
      .
    </Request Type>
  </Operation>
</Response>
```

Request Type Tag and Namespaces

The request type tag must follow the operation type tag within a native data operation request.

Table 4-1 describes the type of the request that must be specified as applying to one of the namespaces.

Table 4-1 Namespace Descriptions

Namespace	Description
<Configuration>	Provides access to the router configuration data analogous to command-line interface (CLI) configuration commands. The allowed operations on configuration data are <Get>, <Set>, and <Delete>.
<Operational>	Provides access to the router operational data and is analogous to CLI show commands. The only operation allowed on operational data is <Get>.
<Action>	Provides access to the action data, for example, the clear commands. The only allowed operation on action data is <Set>.
<AdminOperational>	Provides access to the router administration operational data. The only operation allowed on administration operational data is <Get>.
<AdminAction>	Provides access to the router administration action data, for example, the clear commands. The only allowed operation on administration action data is <Set>.

Object Hierarchy

A hierarchy of elements is included to specify the items to get, set, or delete, and so on, after the request type tag is specified. The precise hierarchy is defined by the XML component schemas.



Note

You should use only the supported XML schema objects; therefore, do not attempt to write a request for other objects.

The XML schema information is mapped to the XML instance.

R3.3 Beta Draft—Cisco Confidential Information

Main Hierarchy Structure

The main structure of the hierarchy consists of the native data model organized as a tree of nodes, where related data items appear in the same branch of the tree. At each level of the tree, a node is a container of further, more specific sets of related data, or a leaf that holds an actual value.

For example, the first element in the configuration data model is `<Configuration>`, which contains all possible configuration items. The children of this element are more specific groups of configuration, such as `<BGP>` for Border Gateway Protocol (BGP) configuration, and `<ISIS>` for Intermediate System-to-Intermediate System (ISIS) configuration. Beneath the `<BGP>` element the data is further compartmentalized with the `<Global>` element for global BGP configuration and `<BGPEntity>` element for per-entity BGP configuration. This compartmentalization continues down to the elements that hold the values, the values being the character data of the element.

The following example shows the main hierarchy structure:

```
<Configuration>
  <BGP>
    .
    .
    .
    <Global>
      .
      .
      .
      <DefaultMetric>10</DefaultMetric>
      .
      .
      .
    </Global>
    <BGPEntity>
      .
      .
      .
    </BGPEntity>
    .
    .
    .
  </BGP>
  <ISIS>
    .
    .
    .
  </ISIS>
</Configuration>
```

Data can be retrieved at any level in the hierarchy—one particular data item can be examined, or all of the data items in a branch of the tree can be returned in one request (see [Chapter 5, “Cisco XML and Native Data Access Techniques,”](#) for details on how to do perform hierarchical data retrieval).

Similarly, configuration data can be deleted at any granularity—one item can be deleted, or a whole branch of related configuration can be deleted. So, for example, all BGP configuration can be deleted in one request, or just the value of the default metric.

Hierarchy Tables

One special type of container element is a table. Tables can hold any number of keyed entries, and are used when there can be multiple instances of an entity. For example, BGP has a table of multiple neighbors, each of which has a unique IP address "key" to identify it. In this case, the table element is

R3.3 Beta Draft—Cisco Confidential Information

<NeighborTable>, and its child element signifying a particular neighbor is <Neighbor>. To specify the key, an extension to the basic parent-child hierarchy is used, where a <Naming> element appears under the child element, containing the key to the table entry.

The following example shows hierarchy tables:

```
<Configuration>
  <BGP>
    .
    .
    .
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <IPAddress>
              <IPv4Address>10.0.101.6</IPv4Address>
            </IPAddress>
          </Naming>
          <RemoteAS>6</RemoteAS>
        </Neighbor>
        <Neighbor>
          <Naming>
            <IPAddress>
              <IPv4Address>10.0.101.7</IPv4Address>
            </IPAddress>
          </Naming>
          <RemoteAS>7</RemoteAS>
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
    .
    .
    .
  </BGP>
  <ISIS>
    .
    .
    .
  </ISIS>
</Configuration>
```

Use tables to access a specific data item for an entry (for example, getting the remote autonomous system number for neighbor 10.0.101.6), or all data for an entry, or even all data for all entries.

Tables also provide the extra feature of allowing the list of entries in the table to be returned. See [“XML Request Using Operation Scope \(Content Attribute\)”](#) section on page 5-82 in Chapter 5, “Cisco XML and Native Data Access Techniques.”

Returned entries from tables can be used to show all neighbors configured, for example, without showing all of their data.

Tables in the operational data model often have a further feature when retrieving their entries. The tables can be filtered on particular criteria to return just the set of entries that fulfill those criteria. For instance, the table of BGP neighbors can be filtered on address family or autonomous system number or update group, or all three. To apply a filter to a table, use another extension to the basic parent-child hierarchy, where a <Filter> element appears under the table element, containing the criteria to filter on.

The following example shows table filtering:

```
<Operational>
  <BGP MajorVersion="15" MinorVersion="0">
    <VRFTTable>
      <VRF>
```

R3.3 Beta Draft—Cisco Confidential Information

```

    <Naming>
      <VrfName>one</VrfName>
    </Naming>
    <NeighborTable>
      <Filter>
        <BGP_AFFilter>
          <AF>IPv4Unicast</AF>
        </BGP_AFFilter>
      </Filter>
    </NeighborTable>
  </VRF>
</VRFTable>
</BGP>
</Operational>

```

Leaf Nodes

The leaf nodes hold values and are generally simple one-value items where the element representing the leaf node uses character data to specify the value (as in "<DefaultMetric>10</DefaultMetric>" in the example in the [“Main Hierarchy Structure”](#) section on page 4-61. In some cases there may be more than one value to specify—for example, when you configure the administrative distance for an address family (the <Distance> element), three values must be given together. Specifying more than one value is achieved by adding further child elements to the leaf, each of which indicates the particular value being configured.

The following example shows leaf nodes:

```

<Configuration>
  <BGP>
    .
    .
    .
    <Distance>
      <ExternalRoutes>20</ExternalRoutes>
      <InternalRoutes>250</InternalRoutes>
      <LocalRoutes>200</LocalRoutes>
    </Distance>
    .
    .
    .
  </BGP>
</Configuration>

```

Sometimes there may be even more structure to the values (with additional levels in the hierarchy beneath the <Distance> tag as a means for grouping the related parts of the data together), although they are still only “settable” or “gettable” as one entity. The extreme example of this is that in some of the information returned from the operational data model, all of the values pertaining to the status of a particular object may be grouped as one leaf. For example, a request to retrieve a particular BGP path status returns all the values associated with that path.

Dependencies Between Configuration Items

Dependencies between configuration items are not articulated in the XML schema nor are they enforced by the XML infrastructure; for example, if item A is this value, then item B must be one of these values, and so forth. The back-end for the Cisco IOS XR applications is responsible for preventing inconsistent configuration from being set. In addition, the management agents are responsible for carrying out the appropriate operations on dependent configuration items through the XML interface.

R3.3 Beta Draft—Cisco Confidential Information

Null Value Representations

The standard attribute “xsi:nil” is used with a value of “true” when a null value is specified for an element in an XML request or response document.

The following example shows how to specify a null value for the element <HoldTime>:

```
<Neighbor>
  <Timers>
    <KeepAlive>60</KeepAlive>
    <HoldTime xsi:nil="true"/>
  </Timers>
</Neighbor>
```

Any element that can be set to “nil” in an XML instance has the attribute “nillable” set to “true” in the XML schema definition for that element. For example:

```
<xsd:element name="HoldTime" type="xsd:unsignedInt" nillable="true"/>
```

Any XML instance document that uses the nil mechanism must declare the “XML Schema for Instance Documents” namespace, which contains the “xsi:nil” definition. Responses to native data operations returned from the router declares the namespace in the operation tag. For example:

```
<Get xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Operation Triggering

When structuring an XML request, the user should remember the following general rule regarding what to specify in the XML for an operation to take place: As a client XML request is parsed by the router, the specified operation takes place whenever a closing tag is encountered after a series of one or more opening tags (but only when the closing tag is not the </Naming> tag).

The following example shows a request to get the BGP timer values for a particular BGP autonomous system. In this example, the <Get> operation is triggered when the <GlobalTimers/> tag is encountered.

Sample XML Client Request to Trigger a <Get> Operation for BGP Timer Values

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <Global>
              <GlobalTimers/>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
```


R3.3 Beta Draft—Cisco Confidential Information

```

<Get>
  <Configuration>
    <BGP MajorVersion="18" MinorVersion="0">
      <AS>
        <Naming>
          <AS>1</AS>
        </Naming>
        <DefaultVRF>
          <Global>
            <ConfederationPeerASTable>
              <ConfederationPeerAS>
                <Naming>
                  <Number>
                    65002
                  </Number>
                </Naming>
                <True>true</True>
              </ConfederationPeerAS>
            </ConfederationPeerASTable>
          </Global>
        </DefaultVRF>
      </AS>
    </BGP>
  </Configuration>
</Get>
</Response>

```

Native Data Operation Examples

These sections provide examples of the basic <Set>, <Get>, and <Delete> operations:

- [Set Configuration Data Request: Example, page 4-65](#)
- [Get Request: Example, page 4-66](#)
- [Get Request of Nonexistent Data: Example, page 4-68](#)
- [Delete Request: Example, page 4-69](#)

Set Configuration Data Request: Example

The following example shows a native data request to set several configuration values for a particular BGP neighbor. Because the <Set> operation in this example is successful, the response contains only the <Set> operation and <Configuration> request type tags.

This request is equivalent to the following CLI commands:

```

router bgp 3
  neighbor 10.0.101.6
    remote-as 6
    ebgp-multihop 255
    address-family ipv4 unicast
      prefix-list orf in
      capability orf prefix-list both
    exit
  address-family ipv4 multicast
    prefix-list orf in
  exit
exit
exit

```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Client Request to <Set> Configuration Values for a BGP Neighbor

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>
                      <IPv4Address>10.0.101.6</IPv4Address>
                    </IPAddress>
                  </Naming>
                  <RemoteAS>6</RemoteAS>
                  <EBGPMultihopMaxHopCount>255</EBGPMultihopMaxHopCount>
                  <NeighborAFTable>
                    <NeighborAF>
                      <Naming>
                        <AF>IPv4Unicast</AF>
                      </Naming>
                      <Activate>true</Activate>
                      <PrefixListFilterIn>orf</PrefixListFilterIn>
                      <AdvertiseORF>Both</AdvertiseORF>
                    </NeighborAF>
                    <NeighborAF>
                      <Naming>
                        <AF>IPv4Multicast</AF>
                      </Naming>
                      <Activate>true</Activate>
                      <PrefixListFilterIn>orf</PrefixListFilterIn>
                    </NeighborAF>
                  </NeighborAFTable>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
</Response>
```

Get Request: Example

The following example shows a native data request to get the address independent configuration values for a specified BGP neighbor (using the same values set in the previous example).

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Client Request to <Get> Configuration Values for a BGP Neighbor

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>
                      <IPv4Address>10.0.101.6</IPv4Address>
                    </IPAddress>
                  </Naming>
                  <RemoteAS>6</RemoteAS>
                  <EBGPMultiHopMaxHopCount>255</EBGPMultiHopMaxHopCount>
                  <NeighborAFTable>
                    <NeighborAF>
                      <Naming>
                        <AF>IPv4Unicast</AF>
                      </Naming>
                      <Activate>true</Activate>
                      <PrefixListFilterIn>orf</PrefixListFilterIn>
                      <AdvertiseORF>Both</AdvertiseORF>
                    </NeighborAF>
                    <NeighborAF>
                      <Naming>
                        <AF>IPv4Multicast</AF>
                      </Naming>
                      <Activate>true</Activate>
                    </NeighborAF>
                  </NeighborAFTable>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

```

        <PrefixListFilterIn>orf</PrefixListFilterIn>
      </NeighborAF>
    </NeighborAFTable>
  </Neighbor>
</NeighborTable>
</BGPEntity>
</DefaultVRF>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

Get Request of Nonexistent Data: Example

The following example shows a native data request to get the configuration values for a particular BGP neighbor similar to the previous example. However, in this example the client application is requesting the configuration for a nonexistent neighbor. Instead of returning an error, the router returns the requested object class hierarchy, but without any data.



Note

Whenever an application attempts to get nonexistent data, the router does not treat this as an error and returns the empty object hierarchy in the response.

Sample XML Client Request to <Get> Configuration Data for a Nonexistent BGP Neighbor

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>
                      <IPV4Address>10.0.101.99</IPV4Address>
                    </IPAddress>
                  </Naming>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">

```

R3.3 Beta Draft—Cisco Confidential Information

```

<AS>
  <Naming>
    <AS>3</AS>
  </Naming>
  <DefaultVRF>
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <IPAddress>
              <IPv4Address>10.0.101.99</IPv4Address>
            </IPAddress>
          </Naming>
          .
          .
          .
          no data returned
          .
          .
          .
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </DefaultVRF>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

Delete Request: Example

The following example shows a native data request to delete the address-independent configuration for a particular BGP neighbor. Note that if a request is made to delete an item that does not exist in the current configuration, an error is not returned to the client application. So in the following example, the returned result is the same as in the previous example: the empty <Delete/> tag, whether or not the specified BGP neighbor exists.

This request is equivalent to the following CLI commands:

```

router bgp 3
  no neighbor 10.0.101.9
exit

```

Sample XML Client Request to <Delete> the Address-Independent Configuration Data for a BGP Neighbor

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Delete>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <IPAddress>

```

R3.3 Beta Draft—Cisco Confidential Information

```

        <IPV4Address>10.0.101.9</IPV4Address>
      </IPAddress>
    </Naming>
  </Neighbor>
</NeighborTable>
</BGPEntity>
</DefaultVRF>
</AS>
</BGP>
</Configuration>
</Delete>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Delete>
    <Configuration/>
  </Delete>
</Response>

```

Cisco XML and Native Data Access Techniques

Beta Draft Cisco Confidential Information

This chapter describes the various techniques or strategies you can use to structure native data operation requests to access the information needed within the extensible markup language (XML) schema object class hierarchy.

Available Set of Native Data Access Techniques

The available native data access techniques are as follows:

- Request all data in the configuration hierarchy. See [“XML Request for All Configuration Data” section on page 5-72.](#)
- Request all configuration data for a component. See [“XML Request for All Configuration Data per Component” section on page 5-72.](#)
- Request all data within a container. See [“XML Request for Specific Data Items” section on page 5-74.](#)
- Combine object class hierarchies within a request. See [“XML Request with Combined Object Class Hierarchies” section on page 5-75.](#)
- Use wildcards in order to apply an operation to a set of entries within a table (Match attribute). See [“XML Request Using Wildcarding \(Match Attribute\)” section on page 5-78.](#)
- Repeat naming information in order to apply an operation to multiple instances of an object. See [“XML Request for Specific Object Instances \(Repeated Naming Information\)” section on page 5-80.](#)
- Perform a one-level <Get> in order to “list” the naming information for each entry within a table (Content attribute). See [“XML Request Using Operation Scope \(Content Attribute\)” section on page 5-82.](#)
- Specify the maximum number of table entries to be returned in a response (Count attribute). See [“Limiting the Number of Table Entries Returned \(Count Attribute\)” section on page 5-84.](#)
- Use custom filters to filter table entries (Filter element). See [“Custom Filtering \(Filter Element\)” section on page 5-85.](#)

The actual data returned in a <Get> request depends on the value of the Source attribute as defined in the [“Getting Configuration Data” section on page 2-30.](#)

R3.3 Beta Draft—Cisco Confidential Information



Note

The term “container” is used in this document as a general reference to any grouping of related data, for example, all of the configuration data for a particular Border Gateway Protocol (BGP) neighbor. The term “table” is used more specifically to denote a type of container that holds a list of named homogeneous objects. For example, the BGP neighbor address table contains a list of neighbor addresses, each of which is identified by its IP address. All table entries in the XML API are identified by the unique value of their <Naming> element.

XML Request for All Configuration Data

Use the empty <Configuration/> tag to retrieve the entire configuration object class hierarchy.

The following example shows how to get the entire configuration hierarchy by specifying the empty <Configuration/> tag.

Sample XML Client Request to <Get> the Entire Configuration Object Class Hierarchy

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration/>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      .
      .
      .
      response data goes here
      .
      .
      .
    </Configuration>
  </Get>
</Response>
```

XML Request for All Configuration Data per Component

All the configuration data for a component is retrieved by specifying the highest level tag for the component.

In the following example, all the configuration data for BGP is retrieved by specifying the empty <BGP/> tag.

Sample XML Client Request for All BGP Configuration Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0"/>
    </Configuration>
  </Get>
</Request>
```


R3.3 Beta Draft—Cisco Confidential Information

```
</Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="18" MinorVersion="0">
        .
        .
        .
        response data goes here
        .
        .
        .
      </BGP>
    </Configuration>
  </Get>
</Response>
```

XML Request for All Data Within a Container

All data within a containers is retrieved by specifying the configuration or operational object class hierarchy down to the containers of interest, including any naming information as appropriate.

The following example shows how to retrieve the configuration for the BGP neighbor with address 10.0.101.6:

Sample XML Client Request to Get All Address Family-Independent Configuration Data Within a BGP Neighbor Container

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>
                  </IPAddress>
                </Naming>
                <RemoteAS>6</RemoteAS>
                <EBGPMultiHopMaxHopCount>255</EBGPMultiHopMaxHopCount>
                <NeighborAFTable>
                  <NeighborAF>
                    <Naming>
                      <AF>IPv4Unicast</AF>
                    </Naming>
                    <Activate>true</Activate>
                    <PrefixListFilterIn>orf</PrefixListFilterIn>
                    <AdvertiseORF>Both</AdvertiseORF>
                  </NeighborAF>
                  <NeighborAF>
                    <Naming>
                      <AF>IPv4Multicast</AF>
                    </Naming>
                    <Activate>true</Activate>
                    <PrefixListFilterIn>orf</PrefixListFilterIn>
                  </NeighborAF>
                </NeighborAFTable>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

XML Request for Specific Data Items

The value of a specific data item (leaf object) can be retrieved by specifying the configuration or operational object class hierarchy down to the item of interest, including any naming information as appropriate.

The following example shows how to retrieve the values of the two data items <RemoteAS> and <EBGPMultiHopMaxHopCount> for the BGP neighbor with address 10.0.101.6:

Sample XML Client Request for Two Specific Data Items: RemoteAS and EBGPMultiHopMaxHopCount

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
```

R3.3 Beta Draft—Cisco Confidential Information

```

<BGP MajorVersion="1" MinorVersion="0">
  <AS>
    <Naming>
      <AS>3</AS>
    </Naming>
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <IPAddress>
              <IPv4Address>10.0.101.6</IPv4Address>
            </IPAddress>
          </Naming>
          <RemoteAS/>
          <EBGPMultihopMaxHopCount/>
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </AS>
</BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>
                  </IPAddress>
                </Naming>
                <RemoteAS>6</RemoteAS>
                <EBGPMultihopMaxHopCount>255</EBGPMultihopMaxHopCount>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>

```

XML Request with Combined Object Class Hierarchies

Multiple object class hierarchies can be specified in a request. For example, a portion of the hierarchy can be repeated, and multiple instances of a child object class can be included under a parent.

R3.3 Beta Draft—Cisco Confidential Information

The object class hierarchy may also be compressed into the most “efficient” XML. In other words, it is not necessary to repeat hierarchies within a request.

Before combining multiple operations inside one <Get> tag, the following limitations should be noted for Release 3.0. Any operations that request multiple items of data must be sent in a separate XML request. They include:

- An operation to retrieve all data beneath a container. For more information, see [“XML Request for All Data Within a Container” section on page 5-73](#).
- An operation to retrieve the list of entries in a table. For more information, see [“XML Request Using Operation Scope \(Content Attribute\)” section on page 5-82](#).
- An operation which includes a wildcard. For more information, see [“XML Request Using Wildcarding \(Match Attribute\)” section on page 5-78](#).

If an attempt is made to make such an operation followed by another operation within the same request, the following error will be returned:

XML Service Library detected the ‘fatal’ condition. The XML document which led to this response contained a request for a potentially large amount of data, which could return a set of iterators. The document also contained further requests for data, but these must be sent in a separate XML document, in order to ensure that they are serviced.

The error indicates that the operations must be separated out into separate XML requests.

The following two examples illustrate two different object class hierarchies that retrieve the same data: the value of the leaf object <RemoteAS> and <EBGPMultiHopMaxHopCount> for the BGP neighbor with the address 10.0.101.6 and all of the configuration data for the BGP neighbor with the address 10.0.101.7:

Example 1: Verbose Form of a Request Using Duplicated Object Class Hierarchies

Sample XML Client Request for Specific Configuration Data Values

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>
                  </IPAddress>
                </Naming>
                <!-- Gets the following two leaf objects for this neighbor -->
                <RemoteAS/>
                <EBGPMultiHopMaxHopCount/>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Get>
```

R3.3 Beta Draft—Cisco Confidential Information

```

<Configuration>
  <BGP MajorVersion="1" MinorVersion="0">
    <AS>
      <Naming>
        <AS>3</AS>
      </Naming>
      <BGPEntity>
        <NeighborTable>
          <Neighbor>
            <Naming>
              <IPAddress>
                <!-- Gets all configuration data for this neighbor -->
                <IPv4Address>10.0.101.7</IPv4Address>
              </IPAddress>
            </Naming>
          </Neighbor>
        </NeighborTable>
      </BGPEntity>
    </AS>
  </BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      .
      .
      .
      response data returned here for
      neighbor 10.0.101.6
      .
      .
    </Configuration>
  </Get>
  <Get>
    <Configuration>
      .
      .
      .
      response data returned here
      neighbor 10.0.101.7
      .
      .
    </Configuration>
  </Get>
</Response>

```

Example 2: Compact Form of a Request Using Compressed Object Class Hierarchies

Sample XML Client Request

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">

```

R3.3 Beta Draft—Cisco Confidential Information

```

<AS>
  <Naming>
    <AS>3</AS>
  </Naming>
  <BGPEntity>
    <NeighborTable>
      <Neighbor>
        <Naming>
          <IPAddress>
            <IPv4Address>10.0.101.6</IPv4Address>
          </IPAddress>
        </Naming>
        <!-- Gets the following two leaf objects for this neighbor -->
        <RemoteAS/>
        <EBGPMultihopMaxHopCount/>
      </Neighbor>
      <Neighbor>
        <Naming>
          <IPAddress>
            <!-- Gets all configuration data for this neighbor -->
            <IPv4Address>10.0.101.7</IPv4Address>
          </IPAddress>
        </Naming>
      </Neighbor>
    </NeighborTable>
  </BGPEntity>
</AS>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      .
      .
      .
      response data returned here for both
      neighbors
      .
      .
      .
    </Configuration>
  </Get>
</Response>

```

XML Request Using Wildcarding (Match Attribute)

Wildcarding of naming information is provided by means of the Match attribute. Match="*" can be used on any Naming attribute within a <Get> or <Delete> operation to effectively specify a wildcarded value for that attribute. The operation applies to all instances of the requested objects.

"*" is the only value supported for Match, though other wildcarding or matching specifications may be supported in the future. The Match attribute is comprehensively supported for table entries in the <Configuration> namespace, but in the <Operational> space the limitation currently exists that nonwildcarded naming information cannot appear in the hierarchy below wildcarded naming information.

R3.3 Beta Draft—Cisco Confidential Information



Note

Although partial wildcarding of NodeIDs is not available in XML, each element of the NodeID has to be wildcarded, similar to the support on the CLI of */* as the only wildcards supported for locations.

The following example shows how to use the Match attribute to get the <RemoteAS> value for all configured BGP neighbors.

Sample XML Client Request Using the Match Attribute Wildcarding

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress Match="*" />
                </Naming>
                <RemoteAS/>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.1</IPv4Address>
                  </IPAddress>
                </Naming>
                <RemoteAS>1</RemoteAS>
              </Neighbor>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.2</IPv4Address>
                  </IPAddress>
                </Naming>
                <RemoteAS>2</RemoteAS>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

```

    </Neighbor>
    <Neighbor>
      <Naming>
        <IPAddress>
          <IPv4Address>10.0.101.3</IPv4Address>
        </IPAddress>
      </Naming>
      <RemoteAS>3</RemoteAS>
    </Neighbor>
    .
    .
    .
    data for more neighbors
    returned here
    .
    .
    .
  </NeighborTable>
</BGPEntity>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

XML Request for Specific Object Instances (Repeated Naming Information)

Wildcarding allows the client application to effectively specify all instances of a particular object. Similarly, the client application might have a need to specify only a limited set of instances of an object. Specifying object instances can be done by simply repeating the naming information in the request.

The following example shows how to retrieve the address independent configuration for three different BGP neighbors, that is, the neighbors with addresses 10.0.101.1, 10.0.101.6, and 10.0.101.8, by repeating the naming information, once for each desired instance.

Sample XML Client Request Using Repeated Naming Information for BGP <NeighborAddress> Instances

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.1</IPv4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPv4Address>10.0.101.6</IPv4Address>

```


R3.3 Beta Draft—Cisco Confidential Information

```

        </IPAddress>
      </Naming>
    </Neighbor>
  </NeighborTable>
  <NeighborTable>
    <Neighbor>
      <Naming>
        <IPAddress>
          <IPv4Address>10.0.101.8</IPv4Address>
        </IPAddress>
      </Naming>
    </Neighbor>
  </NeighborTable>
</BGPEntity>
</AS>
</BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
        <BGPEntity>
          <NeighborTable>
            <Neighbor>
              <Naming>
                <IPAddress>
                  <IPv4Address>10.0.101.1</IPv4Address>
                </IPAddress>
              </Naming>
              .
              .
              .
              data returned for 1st neighbor
              .
              .
              .
            </Neighbor>
            <Neighbor>
              <Naming>
                <IPAddress>
                  <IPv4Address>10.0.101.6</IPv4Address>
                </IPAddress>
              </Naming>
              .
              .
              .
              data returned for 2nd neighbor
              .
              .
              .
            </Neighbor>
          <Neighbor>
            <Naming>
              <IPAddress>

```

R3.3 Beta Draft—Cisco Confidential Information

```

        <IPv4Address>10.0.101.6</IPv4Address>
      </IPAddress>
    </Naming>
    .
    .
    .
    data returned for 3rd neighbor
    .
    .
    .
  </Neighbor>
</NeighborTable>
</BGPEntity>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

XML Request Using Operation Scope (Content Attribute)

The Content attribute is used on any table element in order to specify the scope of a <Get> operation. [Table 5-1](#) describes the content attribute values are supported.

Table 5-1 Content Attributes

Content Attribute	Description
All	Use to get all leaf items and their values. All is the default when the Content attribute is not specified on a table element.
Entries	Use to get the Naming information for each entry within a specified table object class. Entries provides a one-level get capability.

If the Content attribute is specified on a nontable element, it is ignored. Note also that the Content and Count attributes can be used together on the same table element.

The following example displays the Content attribute that is used to list all configured BGP neighbors:

Sample XML Client Request Using the All Content Attribute

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable Content="Entries"/>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>

```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable Content="Entries">
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPV4Address>10.0.101.1</IPV4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPV4Address>10.0.101.2</IPV4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPV4Address>10.0.101.3</IPV4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPV4Address>10.0.101.4</IPV4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
              .
              .
              .
              more neighbors returned here
              .
              .
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Limiting the Number of Table Entries Returned (Count Attribute)

The Count attribute is used on any table element within a <Get> operation to specify the maximum number of table entries to be returned in a response. When the Count attribute is specified, the naming information within the request is used to identify the starting point within the table, that is, the first table entry of interest. If no naming information is specified, the response starts at the beginning of the table.

For a table whose entries are containers, the Count attribute can be used only if the Content attribute is also specified with a value of Entries. This restriction does not apply to a table whose children are leaf nodes.

As an alternative to the use of the Count attribute, the XML interface supports the retrieval of large XML responses in blocks through iterators. For more information on iterators, see [Chapter 7, “Cisco XML and Large Data Retrieval \(Iterators\).”](#)

The following example shows how to use the Count attribute to retrieve the configuration information for the first five BGP neighbors starting with the address 10.0.101.1:

Sample XML Client Request Using the Count Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable Count="5">
              <Neighbor>
                <Naming>
                  <IPAddress>
                    <IPV4Address>10.0.101.1</IPV4Address>
                  </IPAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <BGPEntity>
            <NeighborTable Count="5">
              <Neighbor>
                <Naming>
                  <IPAddress>
```

R3.3 Beta Draft—Cisco Confidential Information

```

        <IPV4Address>10.0.101.1</IPV4Address>
      </IPAddress>
    </Naming>
    .
    .
    .
    data for 1st neighbor returned
    here
    .
    .
    .
  </Neighbor>
  <Neighbor>
    <Naming>
      <IPAddress>
        <IPV4Address>10.0.101.2</IPV4Address>
      </IPAddress>
    </Naming>
    .
    .
    .
    data returned for 2nd neighbor
    here
    .
    .
    .
  </Neighbor>
  .
  .
  .
  data returned for remaining
  neighbors here
  .
  .
  .
</NeighborTable>
</BGPEntity>
</AS>
</BGP>
</Configuration>
</Get>
</Response>

```

Custom Filtering (Filter Element)

Some of the tables from the operational namespace support the selection of rows of interest based on predefined filtering criteria. Filters can be applied to such tables in order to reduce the number of table entries retrieved in a request.

Client applications specify filtering criteria for such tables by using the <Filter> tag and including the filter specific parameters as defined in the XML schema definition for that table. If no table entries match the specified filter criteria, the response contains the object class hierarchy down to the specified table, but does not include any table entries. The Content attribute can be used with a filter to specify the scope of a <Get> request.

R3.3 Beta Draft—Cisco Confidential Information

In the following example, the filter `<BGP_ASFilter>` is used to retrieve operational information for all neighbors in autonomous system 6:

Sample XML Client Request Using Filtering

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Operational>
      <BGP MajorVersion="1" MinorVersion="0">
        <NeighborTable>
          <Filter>
            <BGP_ASFilter>
              <AS>6</AS>
            </BGP_ASFilter>
          </Filter>
        </NeighborTable>
      </BGP>
    </Operational>
  </Get>
</Request>
```

Sample Filtered XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Operational>
      <BGP MajorVersion="1" MinorVersion="0">
        <NeighborTable>
          <Filter>
            <BGP_ASFilter>
              <AS>6</AS>
            </BGP_ASFilter>
          </Filter>
          <Neighbor>
            .
            .
            .
            data for 1st neighbor returned here
            .
            .
            .
          </Neighbor>
          <Neighbor>
            .
            .
            .
            data for 2nd neighbor returned here
            returned here
            .
            .
            .
          </Neighbor>
            .
            .
            .
            data for remaining neighbors returned
            here
            .
            .
            .
          </NeighborTable>
        </BGP>
```

R3.3 Beta Draft—Cisco Confidential Information

```
</Operational>  
</Get>  
</Response>
```

R3.3 Beta Draft – Cisco Confidential Information

Cisco XML and Encapsulated CLI Operations

Beta Draft Cisco Confidential Information

The extensible markup language (XML) interface for the router provides support for XML encapsulated command-line interface (CLI) commands and responses.

This chapter provides information on XML CLI command tags.

XML CLI Command Tags

A client application can request a CLI command by encoding the text for the command within a pair of `<CLI>` start and `</CLI>` end tags, and `<Configuration>` tags. The router responds with the uninterpreted CLI text result.



Note

XML encapsulated CLI commands use the same target configuration as the corresponding XML operations `<Get>`, `<Set>`, and `<Delete>`.

When used for CLI operations, the `<Configuration>` tag supports the optional `Operation` attribute, which can take one of the values listed in [Table 6-1](#).

Table 6-1 Operational Attribute Values

Operational Attribute Value	Operational Attribute Value Description
Apply	Specifies that the commands should be executed or applied (default).
Help	Gets help on the last command in the list of commands sent in the request. There should not be any empty lines after the last command (because the last command is considered to be the one on the last line)
CommandCompletion	Completes the last keyword of the last command. Apart from not allowing empty lines at the end of the list of commands sent in the request, when this option is used there should not be any white spaces after the partial keyword to be completed.

R3.3 Beta Draft—Cisco Confidential Information

The following example uses the <CLI> operation tag:

Sample XML Client Request for CLI Command Using CLI Tags

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <CLI>
    <Configuration>
      router bgp 3
      default-metric 10
      timers bgp 80 160
      exit
      show config
    </Configuration>
  </CLI>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <CLI>
    <Configuration>
      Building configuration...
      router bgp 3
      default-metric 10
      timers bgp 80 160
      end
    </Configuration>
  </CLI>
</Response>
```

CLI Command Limitations

The initial CLI command support through XML is limited to CLI configuration and subsequent responses wrapped in <CLI> tags.

The following commands and conditions are not supported:

- <Operational> namespace commands.
- <Action> namespace commands.
- Iterators for responses to <CLI> commands issued through XML. For example, iterators are not supported for the output of the **show run** and **show configuration** commands.
- Sending a request in <CLI> format and getting back an XML encoded response.
- Sending an XML encoded request and getting back a response in <CLI> format.
- “Long running” commands, for example, “ping” and “top.”
- Iterators in responses to CLI commands such as the **show run** and **show configuration** commands.
- Only one XML <CLI> request can be issued at a time across all client sessions on the router.

Cisco XML and Large Data Retrieval (Iterators)

Beta Draft Cisco Confidential Information

The extensible markup language (XML) for the router supports the retrieval of large XML responses in blocks (that is, in chunks or sections).

This chapter provides information on large data retrieval. For information on terminating an iterator, see [“Terminating an Iterator” section on page 7-94](#).

When a client application makes a request, the resulting response data size is checked to determine if it is larger than a predetermined block size. If it is not larger, then the complete data is returned in a normal response. However, if the response data is larger than the block size, then the first set of data is returned according to the block size along with an iterator ID included as the value of the `IteratorID` attribute. The client must then send `<GetNext>` requests including the iterator ID until all the data is retrieved. The client application knows that all of the data is retrieved when it receives a response that does not contain an `IteratorID` attribute.

The following points should be noted by the client application when iterators are used:

- The block size is a fixed value specific to each transport mechanisms on the router, that is, the XML agent for Common Object Request Broker Architecture (CORBA) and Secure Shell (SSH) or Telnet. No mechanism is provided for the client application to specify a desired block size.
- The block size refers to the entire XML response, not just the payload portion of the response.
- Large responses are divided based on the requested block size, not on the contents. However, each response is always a complete XML document.
- Requests containing multiple operations are treated as a single entity when the block size and `IteratorID` are applied. As a result, the `IteratorID` is an attribute of the `<Response>` tag, never of an individual operation.
- If the client application sends a request that includes an operation resulting in the need for an iterator to return all of the response data, any further operations contained within that request are rejected. The rejected operations are resent in another request.
- The `IteratorID` is an unsigned 32-bit value that should be treated as opaque data by the client application. Furthermore, the client application should not assume that the `IteratorID` is constant between `<GetNext>` operations.

To reduce memory overhead and avoid memory starvation of the router, the following limitations are placed on the number of allowed iterators:

- 10—Maximum number of iterators allowed at any one time on a given client session.
- 100—Maximum number of iterators allowed at any one time for all client sessions.

R3.3 Beta Draft—Cisco Confidential Information


Note

If a <Get> request is issued that results in an iterated response, it is counted as 1 iterator, regardless of the number of <GetNext> operations required to retrieve all of the response data. For example, a <Get> request may require 10, 100, or more <GetNext> operations to retrieve all of the associated data, but during this process only 1 iterator is being used. Also, an iterator is considered to be in use until all of the response data associated with that iterator (that is, all of the response data associated with the original <Get> request) is retrieved or the iterator is terminated with the Abort attribute.

The following example shows a client request that utilizes an iterator to retrieve all global Border Gateway Protocol (BGP) configuration data for a specified autonomous system:

Sample XML Client Request to Retrieve All BGP Configuration Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global/>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router Containing the First Block of Retrieved Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="1">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            1st block of data returned here
            .
            .
            .
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Second XML Client Request Using the <GetNext> Iterator to Retrieve the Next Block of BGP Configuration Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="1"/>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router Containing the Second Block of Retrieved Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="1">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            2nd block of data returned here
            .
            .
            .
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Third XML Client Request Using the <GetNext> Iterator to Retrieve the Next Block of BGP Configuration Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="1"/>
</Request>
```

Sample XML Response from the Router Containing Third Block of Retrieved Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="1">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            3rd block of data returned here
            .
            .
            .
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Final XML Client Request Using the <GetNext> Iterator to Retrieve the Last Block of BGP Configuration Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="1"/>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Final XML Response from the Router Containing the Final Block of Retrieved Data

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            Final block of data returned here
            .
            .
            .
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Terminating an Iterator

A client application may terminate an iterator without retrieving all of the response data by including an Abort attribute with a value of “true” on the <GetNext> operation. A client application that does not complete or terminate its requests risks running out of iterators.

The following example shows a client request using the Abort attribute to terminate an iterator:

Sample XML Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global/>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="2">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            1st block of data returned here
            .
            .
            .
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```

Sample XML Request Using the Abort Attribute to Terminate an Iterator

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="2" Abort="true"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="2" Abort="true"/>
</Response>
```

R3.3 Beta Draft – Cisco Confidential Information

Cisco XML Security

Beta Draft Cisco Confidential Information

Specific security privileges are required for a client application requesting information from the router. This chapter contains the following sections:

- [Authentication, page 8-97](#)
- [Authorization, page 8-97](#)
- [Retrieving Task Permissions, page 8-98](#)
- [Task Privileges, page 8-99](#)
- [Task Names, page 8-99](#)
- [Authorization Failure, page 8-100](#)

Authentication

User authentication through authentication, authorization, and accounting (AAA) is handled on the router by the transport-specific XML agent and is not exposed through the XML interface.

Authorization

Every operation request by a client application is authorized. If the client is not authorized to perform an operation, the operation is not performed by the router and an error is returned.

Authorization of client requests is handled through the standard AAA “task permissions” mechanism. The XML agent caches the AAA user credentials obtained from the user authentication process, and then each client provides these to the XML infrastructure on the router. As a result, no AAA information needs to be passed in the XML request from the client application.

Each object class in the schema has a task ID associated with it. A client application’s capabilities and privileges in terms of task IDs are exposed by AAA through a **show** command. A client application can use the XML interface to retrieve the capabilities prior to sending configuration requests to the router.

A client application requesting an operation through the XML interface must have the appropriate task privileges enabled/assigned for any objects accessed in the operation:

- <Get> operations require AAA “read” privileges.
- <Set> and <Delete> operations require AAA “write” privileges.

R3.3 Beta Draft—Cisco Confidential Information

The “configuration services” operations through configuration manager can also require the appropriate predefined task privileges.

If an operation requested by a client application fails authorization, an appropriate <Error> element is returned in the response sent to the client. For “native data” operations, the <Error> element is associated with the specific element or object classes where the authorization error occurred.

Retrieving Task Permissions

A client application’s capabilities and privileges in terms of task permissions are exposed by AAA through command-line interface (CLI) **show** commands. A client application can also use the XML interface to programmatically retrieve the current AAA capabilities from the router. This retrieval can be done by issuing the appropriate <Get> request to the <AAA> component.

The following example shows a request to retrieve all of the AAA configuration from the router:

Sample XLM Request to Retrieve AAA Configuration Information

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <AAA MajorVersion="1" MinorVersion="0"/>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <AAA MajorVersion="1" MinorVersion="0">
        .
        .
        .
        AAA configuration returned here
        .
        .
        .
      </AAA>
    </Configuration>
  </Get>
</Response>
```

R3.3 Beta Draft—Cisco Confidential Information

Task Privileges

A client application requesting a native data operation through the XML interface must have the appropriate task privileges enabled/assigned for any items accessed in the operation, as follows:

- <Get>, <GetNext>, and <GetVersionInfo> operations require AAA “read” privileges.
- <Set> and <Delete> operations require AAA “write” privileges.

The “configuration services” operations through configuration manager can also require the appropriate predefined task privileges.

Task Names

Each object (that is, data item or table) exposed through the XML interface and accessible to the client application has one or more task names associated with it. The task names are published in the XML schema documents as <appinfo> annotations.

For example, the complex type definition for the top-level element in the Border Gateway Protocol (BGP) configuration schema contains the following annotation:

```
<xsd:appinfo>
  <MajorVersion>1</MajorVersion>
  <MinorVersion>0</MinorVersion>
  <TaskIdInfo TaskGrouping="Single">
    <TaskName>bgp</TaskName>
  </TaskIdInfo>
</xsd:appinfo>
```

Here is another example from a different component schema. This annotation includes a list of task names.

```
<xsd:appinfo>
  <MajorVersion>1</MajorVersion>
  <MinorVersion>0</MinorVersion>
  <TaskIdInfo TaskGrouping="And">
    <TaskName>ouni</TaskName>
    <TaskName>mpls-te</TaskName>
  </TaskIdInfo>
</xsd:appinfo>
```

Task names indicate what permissions are required to access the data below the object. In the example, the task names `ouni` and `mpls-te` are specified for the object. The task names apply to the object and are inherited by all of the descendants of the object in the schema. In other words, the task names that apply to a particular object are the task names specified for the object and the task names of all ancestors for which there is a task name specified in the schema.

The TaskGrouping attribute specifies the logical relationship among the task names when multiple task names are specified for a particular object. For example, for a client application to issue a <Get> request for the object containing the preceding annotation, the corresponding AAA user credentials must have read permissions set for both the `ouni` and `mpls-te` tasks (and any tasks inherited by the object). The possible values for the TaskGrouping attribute are And, Or, and Single. The value Single is used when there is only a single task name specified for the object.

R3.3 Beta Draft – Cisco Confidential Information

Authorization Failure

If an operation requested by a client application fails authorization, an appropriate `<Error>` element is returned in the response sent to the client. For “native data” operations, the `<Error>` element is associated with the specific element or object where the authorization error occurred.

If a client application issues a `<Get>` request to retrieve all data below a container object, and if any subsections of that data require permissions that the user does not have, then an error is not returned. Instead, the subsection of data is not included in the `<Get>` response.

Cisco XML Schema Versioning

Beta Draft Cisco Confidential Information

Before the router can carry out a client application request, it must verify version compatibility between the client request and router component versions.

A major and minor version number are carried on the <Request> and <Response> elements to indicate the overall extensible markup language (XML) application programming interface (API) version in use by the client application and router. In addition, each component XML schema exposed through the XML API has a major and minor version number associated with it.

This chapter describes the format of the version information exchanged between the client application and the router, and how the router uses this information at run time to check version compatibility.

This chapter contains the following sections:

- [Major and Minor Version Numbers, page 9-101](#)
- [Run-Time Use of Version Information, page 9-102](#)
- [Retrieving Version Information, page 9-105](#)

Major and Minor Version Numbers

The top-level or root object (that is, element) in each component XML schema carries the major and minor version numbers for that schema. A minor version change is defined as an addition to the XML schema. All other changes, including deletions and semantic changes, are considered major version changes.

The version numbers are documented in the header comment contained in the XML schema file. They are also available as <xsd:appinfo> annotations included as part of the complex type definition for the top-level schema element. This enables you to programmatically extract the version numbers from the XML schema file to include in XML request instances sent to the router. The version numbers are carried in the XML instances using the MajorVersion and MinorVersion attributes.

R3.3 Beta Draft—Cisco Confidential Information

The following example shows the relevant portion of the complex type definition for an element that carries version information:

```
<xsd:complexType name="ipv4_bgp_cfg_BGP_type">
  <xsd:annotation>
    <xsd:documentation>Global BGP config</xsd:documentation>
    <xsd:appinfo>
      <MajorVersion>1</MajorVersion>
      <MinorVersion>0</MinorVersion>
      <TaskIdInfo TaskGrouping="Single">
        <TaskName>bgp</TaskName>
      </TaskIdInfo>
    </xsd:appinfo>
  </xsd:annotation>
  .
  .
  .
  <xsd:attributeGroup ref="VersionAttributeGroup"/>
  .
  .
  ..
</xsd:complexType>
```

The attribute group VersionAttributeGroup is defined as follows:

```
<xsd:attributeGroup name="VersionAttributeGroup">
  <xsd:annotation>
    <xsd:documentation>
      Common version information attributes
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="MajorVersion" type="xsd:unsignedInt" use="required"/>
  <xsd:attribute name="MinorVersion" type="xsd:unsignedInt" use="required"/>
</xsd:attributeGroup>
```

Run-Time Use of Version Information

Each XML request must contain the client's major and minor version numbers at the appropriate locations in the XML. These version numbers are compared to the version numbers running on the router. The request is then accepted or rejected based on the following rules:

- If there is a major version discrepancy, then the request fails.
- If there is a minor version lag, that is, the client minor version is behind that of the router, then the request is attempted.
- If there is a minor version creep, that is, the client minor version is ahead of that of the router, then the request fails.
- If the version information has not been included in the request, then the request fails.

Each XML response can also contain the version numbers at the appropriate locations in the XML.



Note

If the client minor version is behind that of the router, then the response may contain elements that are not recognized by the client application. The client application must be able to handle these additional elements.

R3.3 Beta Draft—Cisco Confidential Information

Placement of Version Information

The following example shows the placement of the MajorVersion and MinorVersion attributes within a client request to retrieve the global BGP configuration data for a specified autonomous system:

Sample Client Request Showing Placement of Version Information

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global/>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            data returned here
            .
            .
            .
          </Global>
        <AS>
      </BGP>
    </Configuration>
  <Get>
</Response>
```

Version Lag

The following example shows a request and response with a version mismatch. In this case, the client minor version is behind that of the router, so the request is attempted.



Note

The version number, which is returned in the response, is the version number running on the router.

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Client Request with a Version Mismatch

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global/>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="1">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="1">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            .
            .
            .
            data returned here
            .
            .
            .
          </Global>
        <AS>
          </BGP>
        </Configuration>
      </Get>
    </Response>
```

Version Creep

The following example shows a request and response with a version mismatch. In this case, the client minor version is ahead of that of the router minor version, resulting in an error response.

Sample XML Request with a Minor Version Mismatch Ahead of the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="1"/>
    </Configuration>
  </Get>
</Request>
```


R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="12345678">
  <Get xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x43679000"
    ErrorMsg="&apos;XML Service Library&apos; detected the &apos;warning&apos;
    condition &apos;An error was encountered in the XML beneath this operation
    tag&apos;" >
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0" ErrorCode="0x4368ac00"
        ErrorMsg="&apos; XMLMDA&apos; detected the &apos;warning&apos; condition
        &apos; The XML version specified in the XML request is not compatible
        with the version running on the router&apos;"/>
    </Configuration>
  </Get>
</Response>
```

Retrieving Version Information

The version of the XML schemas running on the router can be retrieved using the `<GetVersionInfo>` tag followed by the appropriate tags identifying the names of the desired components.

In the following example, the `<GetVersionInfo>` tag is used to retrieve the major and minor version numbers for the BGP component configuration schema:

Sample XML Request to Retrieve Major and Minor Version Numbers

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetVersionInfo>
    <Configuration>
      <BGP/>
    </Configuration>
  </GetVersionInfo>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetVersionInfo>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0"/>
    </Configuration>
  </GetVersionInfo>
</Response>
```

The following example shows how to retrieve the version information for all configuration schemas available on the router:

Sample XML Request to Retrieve Version Information for All Configuration Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetVersionInfo>
    <Configuration/>
  </GetVersionInfo>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetVersionInfo>
    <Configuration>
      .
      .
      .
      <MPLS_LSD MajorVersion="1" MinorVersion="0"/>
      <MPLS_TE MajorVersion="1" MinorVersion="0"/>
      <OUNI MajorVersion="1" MinorVersion="0"/>
      <OLM MajorVersion="1" MinorVersion="0"/>
      <BGP MajorVersion="1" MinorVersion="0"/>
      <CDP MajorVersion="1" MinorVersion="1"/>
      <RSVP MajorVersion="1" MinorVersion="0"/>
      .
      .
      .
    <InterfaceConfiguration>
      <CDP MajorVersion="1" MinorVersion="0"/>
      <SONET MajorVersion="1" MinorVersion="2"/>
      <PPP MajorVersion="1" MinorVersion="0">
        <IPCP MajorVersion="1" MinorVersion="0"/>
      </PPP>
      .
      .
      .
    </InterfaceConfiguration>
    .
    .
    .
  </Configuration>
</GetVersionInfo>
</Response>
```

Alarms

Beta Draft Cisco Confidential Information

The Cisco IOS XR XML API supports the registration and receipt of notifications, for example, asynchronous responses such as alarms, over any transport. The system supports alarms and event notifications over XML/SSH.

An asynchronous registration request is followed by a synchronous response and any number of asynchronous responses. If a client wants to stop receiving a particular set of asynchronous responses at a later stage, the client sends a deregistration request.

One type of notification that is supported by the Cisco IOS XR XML API is alarms, for example, syslog messages. The alarms that are received are restricted by a filter, which is specified in the registration request. An alarm registration request is followed by a synchronous response. If successful, the synchronous response contains a `RegistrationID`, which is used by the client to uniquely identify the applicable registration. A client can make many alarm registrations. If a client wants to stop receiving a particular set of alarms at a later stage, the client can send a deregistration request for the relevant `RegistrationID` or all `RegistrationIDs` for the session.

When an asynchronous response is received that contains an alarm, the registration that resulted in the alarm is determined from the `RegistrationID`.

The following sections describe the XML used for every operation:

- [Alarm Registration, page 10-107](#)
- [Alarm Deregistration, page 10-108](#)
- [Alarm Notification, page 10-109](#)

Alarm Registration

Alarm registration and deregistration requests and responses and alarm notifications use the `<Alarm>` operation tag to distinguish them from other types of XML operations. A registration request contains the `<Register>` tag, which is followed by several tags that specify the filter requirement. If registration for all alarms is required, no filter is specified. The following filter criteria are listed:

- `SourceID`
- `Category`
- `Group`
- `Context`
- `Code`

R3.3 Beta Draft—Cisco Confidential Information

- Severity
- BiStateOnly

If a success, the response contains a <Register> tag with a RegistrationID attribute. If a failure, the filter tag that caused the error appears with an error message attribute. The following example shows a registration request to receive all alarms for configuration change, for example, commit notifications:

Sample XML Request from the Client Application

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Alarm>
    <Register>
      <Group>LIBTARCFG</Group>
      <Code>COMMIT</Code>
    </Register>
  </Alarm>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
Response MajorVersion="1" MinorVersion="0">
  <Alarm>
    <Register RegistrationID="123"/>
  </Alarm>
</Response>
```



Note

If a second registration is made with the same filter or if the filters with two registrations overlap, these alarms that match both registrations are received twice. In general, each alarm is received once for each registration which it matches.

If a session ends (for example, the connection is dropped), all registrations are automatically canceled.

Alarm Deregistration

An alarm deregistration request consists of the <Alarm> operation tag followed by the <Deregister> tag, with the optional attribute RegistrationID. If RegistrationID is specified, the value must be that returned from a previous registration request. The registration with that ID must not have already been deregistered or an error is returned. If it is not specified, the request results in all alarm registrations for that session being deregistered.

The following example shows a deregistration request for the RegistrationID returned from the registration request example:

Sample XML Request from the Client Application

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Alarm>
    <Deregister RegistrationID="123"/>
  </Alarm>
</Request>
```

R3.3 Beta Draft—Cisco Confidential Information

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Alarm>
    <Deregister RegistrationID="123"/>
  </Alarm>
</Response>
```

Alarm Notification

Alarm notifications are contained within a pair of <Notification> tags to distinguish them from normal responses. Each notification contains one or more alarms, each of which is contained within a pair of <Alarm> tags. The tags have an attribute RegistrationID, where the value is the RegistrationID returned in the registration that resulted in the alarm.

The tags contain the following fields for the alarm:

- SourceID
- EventID
- Timestamp
- Category
- Group
- Code
- Severity
- State
- CorrelationID
- AdditionalText

The following example shows the configuration commit alarm notification:

```
<?xml version="1.0" encoding="UTF-8"?>
<Notification>
  <Alarm RegistrationID="123">
    <SourceID>RP/0/0/CPU0</SourceID>
    <EventID>84</EventID>
    <Timestamp>1077270612</Timestamp>
    <Category></Category>
    <Group>LIBTARCFG</Group>
    <Code>COMMIT</Code>
    <Severity>Informational</Severity>
    <State>NotAvailable</State>
    <CorrelationID>0</CorrelationID>
    <AdditionalText>config[65704]: %LIBTARCFG-6-COMMIT : Configuration committed
by user &#39;admin&#39;. Use &#39;show commit changes 1000000490&#39; to view
the changes.</AdditionalText>
  </Alarm>
</Notification>
```

R3.3 Beta Draft – Cisco Confidential Information

Error Reporting in Cisco XML Responses

Beta Draft Cisco Confidential Information

The extensible markup language (XML) responses returned by the router contains error information as appropriate, including the operation, object, and cause of the error when possible. The error codes and messages returned from the router may originate in the XML agent or in one of the other infrastructure layers; for example, the XML Service Library, XML Parser Library, or Configuration Manager.

Types of Reported Errors

The types of potential errors in XML Responses are listed in [Table 11-1](#).

Table 11-1 *Reported Error Types*

Error Type	Description
Transport errors	Transport-specific errors are detected within the XML agent (and include failed authentication attempts).
XML parse errors	XML format or syntax errors are detected by the XML Parser Library (and include errors resulting from malformed XML, mismatched XML tags, and so on).
XML schema errors	XML schema errors are detected by the XML operation provider within the infrastructure (and include errors resulting from invalid operation types, invalid object hierarchies, values out of range, and so on).
Operation processing errors	Operation processing errors are errors encountered during the processing of an operation, typically as a result of committing the target configuration (and include errors returned from Configuration Manager and the infrastructure such as failed authorization attempts, and “invalid configuration errors” returned from the back-end Cisco IOS XR applications).

These error categories are described in the following sections:

- [Error Attributes, page 11-112](#)
- [Transport Errors, page 11-112](#)
- [XML Parse Errors, page 11-112](#)

R3.3 Beta Draft—Cisco Confidential Information

- [XML Schema Errors, page 11-113](#)
- [Operation Processing Errors, page 11-115](#)
- [Error Codes and Messages, page 11-115](#)

Error Attributes

If one or more errors occur during the processing of a requested operation, the corresponding XML response includes error information for each element or object class in error. The error information is included in the form of `ErrorCode` and `ErrorMsg` attributes providing a relevant error code and error message respectively.

If one or more errors occur during the processing of an operation, error information is included for each error at the appropriate point in the response. In addition, error attributes are added at the operation element level. As a result, the client application does not have to search through the entire response to determine if an error has occurred. However, the client can still search through the response to identify each of the specific error conditions.

Transport Errors

Transport-specific errors, including failed authentication attempts, are handled by the appropriate XML agent.

XML Parse Errors

This general category of errors includes those resulting from malformed XML and mismatched XML tags.

The router checks each XML request, but does not validate the request against an XML schema. If the XML contains invalid syntax and thus fails the well-formedness check, the error indication is returned in the form of error attributes placed at the appropriate point in the response. In such cases, the response may not contain the same XML as was received in the request, but just the portions to the point where the syntax error was encountered.

In the following example, the client application sends a request to the router that contains mismatched tags, that is, the opening `<BGPEntity>` tag is not paired with a closing `</BGPEntity>` tag. This example illustrates the format and placement of the error attributes.



Note

The actual error codes and messages might be different than what is shown in this example. Also, the actual error attributes does not contain new line characters.

Sample XML Client Request Containing Mismatched Tags

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
```


R3.3 Beta Draft—Cisco Confidential Information

```

        <BGPEntity>
      </AS>
    </BGP>
  </Configuration>
</Set>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x43679000"
    ErrorMsg="&apos;XML Service Library&apos; detected the &apos;warning&apos;
    condition &apos;An error was encountered in the XML beneath this operation tag
    &apos; ">
    <Configuration ErrorCode="0xa240da00" ErrorMsg="&apos;XML Infrastructure &apos;
    detected the &apos;fatal&apos; condition &apos;Opening and ending tag does not
    match&apos;"/>
  </Get>
</Response>

```

XML Schema Errors

XML schema errors are detected by the XML operation providers. This general category of errors includes those resulting from invalid operation types, invalid object hierarchies, and invalid naming or value elements. However, some schema errors may go undetected because, as previously noted, the router does not validate the request against an XML schema.

In the following example, the client application has requested a <Set> operation specifying an object <ExternalRoutes> that does not exist at this location in the Border Gateway Protocol (BGP) component hierarchy. This example illustrates the format and placement of the error attributes.



Note

The actual error codes and messages may be different than those shown in the example.

Sample XML Client Request Specifying an Invalid Object Hierarchy

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            <ExternalRoutes>10</ExternalRoutes>
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x4368a400"

```

R3.3 Beta Draft—Cisco Confidential Information

```

    ErrorMsg="&apos;XML Service Library&apos; detected the &apos;warning&apos;
    condition &apos;An error was encountered in the XML beneath this operation
    tag&apos;";>
  <Configuration>
    <BGP MajorVersion="1" MinorVersion="0"
      <AS>
        <Naming>
          <AS>3</AS>
        </Naming>
        <Global ErrorCode="0x4368a400" ErrorMsg="&apos;XMLMDA&apos; detected the
          &apos;warning&apos; condition &apos;
          The XML request does not conform to the schema. A child element of
          the element on which this error appears is invalid. No such child
          element name exists at this location in the schema. Please check
          the request against the schema.&apos;"/>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>

```

The following example also illustrates a schema error. In this case, the client application has requested a `<Set>` operation specifying a value for the `<GracefulRestartTime>` object that is not within the range of valid values for this item.

Sample XML Request Specifying an Invalid Object Value Range

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            <GracefulRestartTime>6000</GracefulRestartTime>
          </Global>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x4368a800"
    ErrorMsg="&apos;XML Service Library&apos; detected the &apos;warning&apos;
    condition &apos;An error was encountered in the XML beneath this operation
    tag&apos;";>
    <Configuration>
      <BGP MajorVersion="1" MinorVersion="0">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <Global>
            <GracefulRestartTime ErrorCode="0x4368a800" ErrorMsg="&apos;
              XMLMDA&apos; detected the &apos;warning&apos; condition &apos;
              The XML request does not conform to the schema. The character data
              contained in the element on which this error appears (or one of its

```

R3.3 Beta Draft—Cisco Confidential Information

```

        child elements) does not conform to the XML schema for its datatype.
        Please check the request against the schema.'"/>
    </Global>
  </AS>
</BGP>
</Configuration>
</Set>
</Request>

```

Operation Processing Errors

Operation processing errors include errors encountered during the processing of an operation, typically as a result of committing the target configuration after previous <Set> or <Delete> operations. While processing an operation, errors are returned from Configuration Manager and the infrastructure, failed authorization attempts occur, and “invalid configuration errors” are returned from the back-end Cisco IOS XR applications.

The following example illustrates an operation processing error resulting from a <GetNext> request specifying an unrecognized iterator ID:

Sample XML Client Request and Processing Error

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <GetNext IteratorID="1" Abort="true"/>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" ErrorCode="0xa367a800" ErrorMsg="&apos;
XML Service Library&apos; detected the &apos;fatal&apos; condition &apos;The XML
Infrastructure has been provided with an iterator ID which is not recognized. The
iterator is either invalid or has timed out.&apos;"/>

```

For more information on errors resulting from a commit of the target configuration, see [“Commit Errors” section on page 2-38](#).

Error Codes and Messages

The error codes and messages returned from the router may originate in any one of several components.

The error codes (cernos) returned from these layers are 32-bit integer values. In general, for a given error condition, the error message returned in the XML is the same as the error message displayed on the command line interface (CLI).

R3.3 Beta Draft – Cisco Confidential Information

XML Transport and Event Notifications

Beta Draft Cisco Confidential Information

This chapter discusses the Common Object Request Broker Architecture (CORBA) as the extensible markup language (XML) transport mechanism for the router, and its Internet Inter-ORB Protocol (IIOP), the protocol used for accessing objects across the Internet. CORBA-based event notifications are also discussed in this chapter.

The following Object Request Brokers are supported by Cisco IOS XR software for the router system as clients:

- Java ORB
- IONA Orbix (running on Solaris 2.8)

The chapter contains the following sections:

- [Cisco XML Transport and CORBA IDL, page 12-117](#)
- [CORBA NameServer, page 12-120](#)
- [CORBA Notification Structure, page 12-121](#)
- [CORBA XML Agent Initiative, page 12-122](#)
- [CORBA XML Limitations, page 12-122](#)
- [XML Agent Errors, page 12-123](#)
- [TTY-Based Transports, page 12-123](#)

Cisco XML Transport and CORBA IDL

CORBA/IIOP is the XML transport mechanism. External client applications use CORBA Interface Definition Language (IDL) to exchange XML encoded request and response streams with the CORBA XML agent running on the router.

Communication between the client application and the server (that is, through the CORBA XML agent) is through IIOP. The client application must first bind to the CORBA XML agent object by obtaining the appropriate Interoperable object reference (IOR) from the CORBA Naming Service. The client then obtains a login authentication using the “login()” method. After the client has obtained a login authentication, the client can use the “invoke()” method to send XML requests to the CORBA XML agent.

R3.3 Beta Draft—Cisco Confidential Information

When the CORBA XML agent receives the XML request, it uses the XML infrastructure on the router to parse and process the request. The agent then obtains the XML response from the XML infrastructure and passes this back to the client as a response parameter on the “invoke()” method return.

IDL Interface

The following IDL example defines the interfaces used to make XML API requests. The IDL interface was designed to be simple for easy migration to other transports. Request details, including operation name, request parameters, and versioning information, are not exposed through the IDL, but are instead encoded in the request itself. To make an XML API request, the client application calls the “invoke()” method sending the “stringified” XML request as the request parameter. The stringified XML response along with error information is returned in the response parameter to the client application.

```
module Manageability
{
    interface XMLAgent
    {
/*
* edt: * * XMLAgent::login
*
* Provides the definition for the login() method for the XMLAgent idl
*
* Return: CORBA::Boolean
*
* TRUE - The method succeeded
* FALSE - The method failed
*
* Argument: username
*
* IN - valid login username on the router.
*
* Argument: password
*
* IN - valid unencrypted password for the given username on the router.
*
* Argument: session_context
*
* OUT - internally generated context for a given valid login session.
* Clients will use this in the invoke call.
*
* Argument: response
*
* OUT - contains error code and error message string in case login failed.
* Clients will use this to know the reason of failure.
*/
        boolean login (in string username,
                       in string password,
                       out long session_context,
                       out string response);

/*
* edt: * * XMLAgent::invoke
*
* Provides the definition for the invoke() method for the XMLAgent idl
*
* Return: None
*
* Argument: session_context
*/
    }
```

R3.3 Beta Draft—Cisco Confidential Information

```

* IN - internally generated context for a given valid login session.
* Clients pass this as it is obtained in login method.
*
* Argument: request
*
* IN - XML request string. Passed to XML Infra for processing
*
* Argument: response
*
* OUT - contain response obtained from XML Infra or error code
* and error message string in case of response failure from XML Infra.
*
*/
void invoke (in long session_context,
             in string request,
             out string response);

/*
* edt: * * XMLAgent::logout
*
* Provides the definition for the logout() method for the XMLAgent idl
*
* Return: None
*
* Argument: session_context
*
* IN - internally generated context for a given valid login session.
* Clients will pass as it is obtained in login method.
*
* Argument: response
*
* OUT - contain error code and error message string in case of failure.
*
*/
void logout (in long session_context,
             out string response);

};
};

```

Authentication

Client applications must authenticate with the CORBA XML agent before sending any requests. The authentication is done through the “login()” method of the CORBA XML agent IDL. The username and password supplied are used to contact and call authentication, authorization, and accounting (AAA) server APIs for authentication on the router. Only after successful authentication is the client application able to send XML encode requests to the router using the “invoke()” method.

[Table 12-1](#) describes the “login()” implementation in the CORBA XML agent that uses AAA options as part of the Authentication API.

Table 12-1 Authentication, Authorization, and Accounting Options

Name	Description
default	Determines the authentication method.
ASCII authentication	Determines the username and password. As a result, the username and password supplied in the “login()” method call should be unencrypted ASCII text. Any security for transport should use SSL ¹ .

R3.3 Beta Draft—Cisco Confidential Information

1. SSL = secure socket layer

CORBA NameServer

The NameServer is used by client applications to obtain the interface object request (IOR) of the CORBA XML agent object. In order to make the IOR available to clients, the CORBA XML agent exports it to the NameServer upon startup.

The CORBA Naming Service stores a name with each object reference and provides a hierarchical namespace to allow server implementers to logically organize the IORs of the objects that they export. Naming Contexts are CORBA objects within the Naming Service. There are several operations and methods defined in the naming context IDL. The most commonly used of these are "bind()" and "resolve()". Servers export IORs in a particular naming context by using the "bind()" operation, specifying the name of the object being exported along with the corresponding IOR. Clients then use the "resolve()" operation to obtain the IOR corresponding to a particular name.

Client Access to the Root Naming Context

The standard mechanism for finding the Naming Service and obtaining the root Naming Context is the "resolve_initial_references()" API. This API makes use of proprietary mechanisms to contact the Naming Service and extract the root naming context. Generally, this involves having the environment previously configured with the host on which the name server is running, the port ID, and so on.

The client application can connect to the Naming Service running on the router using a Domain Name Server (DNS) configured router name or through the Naming Service IOR as follows:

```
client -ORBInitRef NameService=iioploc://<router_name>:10001/NameService
```

```
client -ORBInitRef NameService=file://<IOR file>
```

If CORBA is used with SSL, then the IOR method of contacting to the Naming Service must be used along with a service configurator file:

```
client -ORBInitRef NameService=file://<IOR file> -ORBSvcConf <config file>
```

Here the service configurator file would contain the SSL variable definitions, for example:

```
static SSLIOP_Factory "-SSLAuthenticate NONE -SSLPrivateKey
                      PEM:server_key.pem -SSLCertificate PEM:server_cert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

The Naming Service IOR file can be obtained from the router through HTTP at:

```
https://<router-name>/cwi/ns_ssl.ior
```

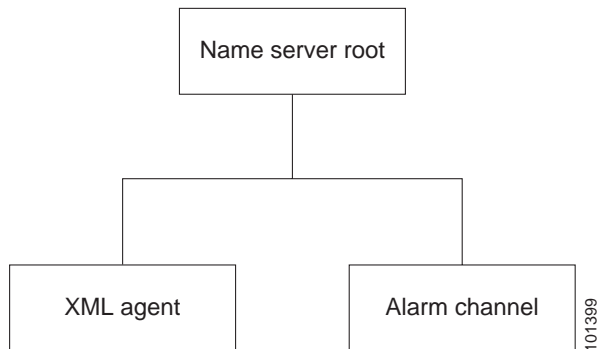


Note

Connection to the Naming Service through an IP address is not supported.

LR Name Server Tree

The name hierarchy for the router is shown in [Figure 12-1](#). The root naming context identifies the particular logical router (LR) on which the Name Server is running. Within the root context is the list of object references that components within the LR have exported.

R3.3 Beta Draft—Cisco Confidential Information**Figure 12-1 Name Server Root**

Event Notifications and Alarms

The CORBA Notification Service is used to deliver events asynchronously to registered clients. Each LR exports one structured notification channel for alarms called “AlarmChannel.” This channel is exported to the Name Server as shown in the Name Server tree for the LR. Client applications can obtain the AlarmChannel IOR and use its standard IDL interface to register interest in events and alarms based on filters.

CORBA Notification Structure

Client applications can receive asynchronous events through the CORBA Notification Service. The alarm notifications are in the form of CORBA structured events where the fixed header consists of the following definitions:

- Domain = LR DNS Name
- Type = alarm
- Instance = Event ID

The complete definition of the structured event is as follows:

Fixed header:

```
<LR_dnsname>.alarm.<event_id>
```

Filterable data:

```
filter:SourceID :
filter:Category :
filter:Severity :
```

Variable header:

```
variable:SourceID :
variable:Category :
variable:Severity :
```

Payload:

```
<?xml version= "1.0" encoding= "UTF-8"?>
<Event Version= "1.0" Module= "AlarmAgent">
```

R3.3 Beta Draft—Cisco Confidential Information

```

<Alarm>
  <SourceID>String</SourceID>
  <EventID>Number</EventID>
  <Timestamp>Number</Timestamp>
  <Category>String</Category>
  <Group>String</Group>
  <Code>String</Code>
  <Severity>String</Severity>
  <State>String</State>
  <CorrelationID>Number</CorrelationID>
  <AdditionalText>String</AdditionalText>
</Alarm>
</Event>

```

Severity and State are enumerations are as follows:

Severity:

```

Unknown,
Emergency,
Alert,
Critical ,
Error ,
Warning,
Notice,
Informational,
Debug

```

State:

```

NotAvailable,
Active,
Clear

```

CORBA XML Agent Initiative

The CORBA XML agent is started on the router through the **xml agent corba** command-line interface (CLI) command. The **ssl** option is used to enable SSL for any subsequent client-to-agent CORBA connections.

The CORBA XML agent is started as follows:

```

RP/0/RPO/CPU0:router# configure terminal
RP/0/RPO/CPU0:router(config)# xml agent corba ssl
RP/0/RPO/CPU0:router(config)# commit
RP/0/RPO/CPU0:router(config)# exit

```

CORBA XML Limitations

The system supports a minimum of 50 simultaneous sessions.

The following limitations apply to the XML transport over CORBA:

- The maximum size of the response data returned to the client application is 32 KB. Responses containing more than 32 KB is returned through iterators as described in [Chapter 7, “Cisco XML and Large Data Retrieval \(Iterators\).”](#)
- The maximum number of simultaneous client logins that is accepted by the CORBA XML agent is five. Each client may in turn have up to four open connections.

R3.3 Beta Draft—Cisco Confidential Information

- The inactivity timeout for a client connection is 15 minutes. After this timeout, the client connection is terminated by the CORBA XML agent.
- Client Object Request Brokers may have buffer limits that need to be adjusted for large XML responses.

XML Agent Errors

All the errors are returned as XML responses with embedded error codes and messages as defined in [Chapter 11, “Error Reporting in Cisco XML Responses.”](#)

[Table 12-2](#) describes the error codes specific to the XML agent for CORBA.

Table 12-2 XML Agent Error Codes

Error Code	Description
0x00000000	Success
0x7FFF0001	Login session failed to authenticate
0x7FFF0002	Login session start failed
0x7FFF0003	Login session activation failed
0x7FFF0004	Login session context invalid
0x7FFF0005	XML response dump from XML infrastructure has failed
0x7FFF0006	XML parsing in XML infrastructure has failed
0x7FFF0007	Login session handle or context is invalid

TTY-Based Transports

These sections describe how to use the TTY-based transports:

- [Enabling the TTY XML Agent, page 12-123](#)
- [Enabling a Session from a Client, page 12-124](#)
- [Sending XML Requests and Receiving Responses, page 12-124](#)
- [Errors That Result in No XML Response Being Produced, page 12-124](#)
- [Ending a Session, page 12-124](#)

Enabling the TTY XML Agent

To enable the TTY agent on the router, which is ready to handle incoming XML sessions over Telnet and Secured Shell (SSH), enter the **xml agent tty** command, as shown in the following example:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# xml agent tty
RP/0/RP0/CPU0:router(config)# commit
RP/0/RP0/CPU0:router(config)# exit
```

For more information about the **xml agent tty** command, see the *Cisco IOS XR System Management Configuration Guide*.

R3.3 Beta Draft—Cisco Confidential Information**Enabling a Session from a Client**

To enable a session from a remote client, invoke SSH or Telnet to establish a connection with the management port on the router. When prompted by the transport protocol, enter a valid username and password. After you have successfully logged on, enter **xml** at the router prompt to be in XML mode.

A maximum of 50 XML sessions can be started over TTY and SSH.

**Note**

You should use, if configured, either the management port or any of the external interfaces rather than a connection to the console or auxiliary port. The management port can have a significantly higher bandwidth and offer better performance.

Sending XML Requests and Receiving Responses

To send an XML request, write the request to the Telnet/SSH session. The session can be used interactively, for example, typing or pasting the XML at the XML> prompt from a window.

**Note**

The XML request must be followed by a new-line character; for example, press **Return**, before the request is processed.

Any responses, either synchronous or asynchronous, are also displayed in the session window. The end of a synchronous response is always represented with </Response> and asynchronous responses (for example), notifications, end with </Notification>.

The client application is single threaded in the context of one session and sends requests synchronously; for example, requests must not be sent until the response to the previous request is received.

Errors That Result in No XML Response Being Produced

If the XML infrastructure is unable to return an XML response, the TTY agent returns an error code and message in the following format:

```
ERROR: 0x%x %s\n
```

Ending a Session

If you are using a session interactively from a terminal window, you can close the window. If you want to manually exit the session, at the prompt:

1. Enter the **exit** command to end XML mode.
2. Enter the **exit** command to end the Telnet/SSH session.

Summary of Cisco XML API Configuration Tags

Beta Draft Cisco Confidential Information

Table 13-1 provides the command-line interface (CLI) to extensible markup language (XML) application programming interface (API) tag mapping for the router target configuration.

Table 13-1 CLI Command or Operation to XML Tag Mapping

CLI Command or Operation	XML Tag
To end, abort, or exit ¹ (from top config mode)	<Unlock> ²
clear	<Clear>
show config	<Get> with <Configuration Source="ChangedConfig">
show config running	<Get> with <Configuration Source="CurrentConfig">
show config merge	<Get> with <Configuration Source="MergedConfig">
show config failed	<Load> with <FailedConfig> followed by <Get> with <Configuration Source="ChangedConfig">
configure exclusive ³	<Lock> ⁴
To change the selected config	<Set> with <Configuration>
To delete the selected config	<Delete> with <Configuration>
commit best-effort	<Commit Mode="BestEffort">
commit	<Commit Mode="Atomic">
show config failed	<Load> with <FailedConfig>
show commit changes commitid	<Get> with <Configuration Source="CommitChanges" ForCommitID=" commitid ">
show commit changes since commitid	<Get> with <Configuration Source="CommitChanges" SinceCommitID=" commitid ">
rollback configuration to commitid	<Rollback> with <CommitID>
rollback configuration last number	<Rollback> with <Previous>
show rollback changes to commitid	<Get> with <Configuration Source="RollbackChanges" ToCommitID=" commitid ">
show rollback changes last number	<Get> with <Configuration Source="RollbackChanges" PreviousCommits=" number ">

R3.3 Beta Draft—Cisco Confidential Information

Table 13-1 ***CLI Command or Operation to XML Tag Mapping (continued)***

CLI Command or Operation	XML Tag
show rollback points	<GetConfigurationHistory RollbackOnly="true">
show configuration sessions	<GetConfigurationSessions>

1. These CLI operations end the configuration session and unlock the running configuration session if it was locked.
2. This XML tag releases the lock on a running configuration but does not end the configuration session.
3. This CLI command starts a new configuration session and locks the running configuration.
4. This XML tag locks the running configuration from a configuration session that is already in progress.

Cisco XML Schemas

Beta Draft Cisco Confidential Information

This chapter contains information about common XML schemas. The structure and allowable content of the extensible markup language (XML) request and response instances supported by the Cisco IOS XR XML application programming interface (API) are documented by means of XML schemas (.xsd files).

The XML schemas are documented using the standard World Wide Web Consortium (W3C) XML schema language, which provides a much more powerful and flexible mechanism for describing schemas than can be achieved using Document Type Definitions (DTDs). The set of XML schemas consists of a small set of common high-level schemas and a larger number of component-specific schemas as described in this chapter.

For more information on the W3C XML Schema standard, see <http://www.w3.org/XML/Schema>

This chapter contains the following sections:

- [XML Schema Retrieval, page 14-127](#)
- [Common XML Schemas, page 14-128](#)
- [Component XML Schemas, page 14-128](#)

XML Schema Retrieval

The XML schemas that belong to the features in a particular package are obtained as a .tar file from cisco.com. To retrieve the XML schemas, you must:

1. Click the following URL to display the IOS XR Software Selector page:
<http://www.cisco.com/cgi-bin/Software/IOXPlanner/planner-tool/ioxplanner.cgi?>



Note Only customer/partner viewers can access the IOS XR Software Selector page. Guest users will get an error.

2. Select the platform from the Select Platform column.
3. Select the XML schema for your platform.

Once untarred, all the XML schema files appear as a flat directory of .xsd files and can be opened with any XML schema viewing application, such as XMLSpy.

R3.3 Beta Draft—Cisco Confidential Information

Common XML Schemas

Among the .xsd files that belong to a BASE package are the common Cisco IOS XR XML schemas that include definitions of the high-level XML request and response instances, operations, and common datatypes. The following common XML schemas are listed:

- alarm_operations.xsd
- config_services_operations.xsd
- cli_operations.xsd
- common_datatypes.xsd
- xml_api_common.xsd
- xml_api_protocol.xsd
- native_data_common.xsd
- native_data_operations.xsd

Component XML Schemas

In addition to the common XML schemas, component XML schemas (such as native data) are provided and contain the data model for each feature. There is typically one component XML schema for each major type of data supported by the component—configuration, operational, action, administration operational, and administration action data—plus any complex data type definitions in the operational space.

**Note**

Sometimes common schema files exist for a component that contain resources used by the component's other schema files (for example, the data types to be used by both configuration data and operational data).

You should use only the XML objects that are defined in the XML schema files. You should not use any unpublished objects that may be shown in the XML returned from the router.

Schema File Organization

There is no hard link from the high-level XML request schemas (namespace_types.xsd) and the component schemas. Instead, links appear in the component schemas in the form of include elements that specify the file in which the parent element exists. The name of the component .xsd file also indicates where in the hierarchy the file's contents reside. If the file ends with _cfg.xsd, it appears as a child of "Configuration"; if it ends with _if_cfg.xsd, it appears as a child of "InterfaceConfiguration", and so on. In addition, the comment header in each .xsd file names the parent object of each top level object in the schema.

R3.3 Beta Draft—Cisco Confidential Information

Schema File Upgrades

If a new version of a schema file becomes available (or has to be uploaded to the router as part of an upgrade), the new version of the file can replace the old version of the file in a straight swap. All other files are unaffected. Therefore, if a component is replaced, only the .xsd files pertaining to that component is replaced.

R3.3 Beta Draft – Cisco Confidential Information

Cisco IOS XR Perl Scripting Toolkit

Beta Draft Cisco Confidential Information

This chapter describes the Cisco IOS XR Perl Scripting Toolkit as an alternative method to the existing management methods. This method enables the router to be managed by a Perl script running on a separate machine. Management commands and data are sent to and from the router in the form of extensible markup language (XML) over either a Telnet or an SSH connection. The well-defined and consistent structure of XML, which is used for both commands and data, makes it easy to write scripts that can interactively manage the router, display information returned from the router in the format required, or manage multiple routers at once.

The following sections describe how to use the Cisco IOS XR Perl Scripting Toolkit:

- [Cisco IOS XR Perl Scripting Toolkit Concepts, page 15-132](#)
- [Security Implications for the Cisco IOS XR Perl Scripting Toolkit, page 15-132](#)
- [Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit, page 15-132](#)
- [Installing the Cisco IOS XR Perl Scripting Toolkit, page 15-133](#)
- [Using the Cisco IOS XR Perl XML API in a Perl Script, page 15-134](#)
- [Handling Types of Errors for the Cisco IOS XR Perl XML API, page 15-134](#)
- [Starting a Management Session on a Router, page 15-134](#)
- [Closing a Management Session on a Router, page 15-136](#)
- [Sending an XML Request to the Router, page 15-136](#)
- [Using Response Objects, page 15-136](#)
- [Using the Error Objects, page 15-137](#)
- [Using the Configuration Services Methods, page 15-138](#)
- [Using the Cisco IOS XR Perl Data Object Interface, page 15-140](#)
- [Cisco IOS XR Perl Notification and Alarm API, page 15-149](#)
- [Examples of Using the Cisco IOS XR Perl XML API, page 15-153](#)

R3.3 Beta Draft—Cisco Confidential Information

Cisco IOS XR Perl Scripting Toolkit Concepts

Table 15-1 describes the toolkit concepts. Some sample scripts are modified and show how to use the API in your own scripts.

Table 15-1 List of Concepts for the IOS XR Perl Scripting Toolkit

Concept	Definition
Cisco IOS XR Perl XML API ¹	Consists of the core of the toolkit and provides the ability to create management sessions, send management requests, and receive responses by using Perl objects and methods.
Cisco IOS XR Perl Data Object API	Allows management requests to be sent and responses received entirely using Perl objects and data structures without any knowledge of the underlying XML.
Cisco IOS XR Perl Notification/Alarm API	Allows a script to register for notifications (for example, alarms), on a management session and receive the notifications asynchronously as Perl objects.

1. API = application programming interface

Security Implications for the Cisco IOS XR Perl Scripting Toolkit

Similar to using the command-line interface (CLI) over a Telnet or Secured Shell (SSH) connection, all authentication and authorization are handled by authentication, authorization, and accounting (AAA) on the router. A script prompts you to enter a password at run time, which ensures that passwords never get stored on the client machine. Therefore, the security implications for using the toolkit are identical to the CLI over the same transport.

Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit

To use the toolkit, you must have installed Perl version 5.6 on the client machine that runs UNIX and Linux. To use the SSH transport option, you must have the SSH client executable installed on the machine and in your path.

You need to install the following specific standard Perl modules to use the various functions:

- **XML::LibXML**—This module is essential for using the Perl XML API and requires that the libxml2 library be installed on the system first, which must be the version that is compatible with the version of XML::LibXML. The toolkit is tested to work with XML::LibXML version 1.58 and libxml2 version 2.6.6. If you are installing libxml2 from a source, you must apply the included patch file before compiling.
- **Term::ReadKey** (optional but recommended)—This module reads passwords without displaying them on the screen.
- **Net::Telnet**—This module is needed if you are using the Telnet or SSH transport modules.

If one of the modules is not available in the current version, you are warned during the installation process. Before installing the toolkit, you should install the current versions of the modules. You can obtain all modules from the following location: <http://www.cpan.org/>

R3.3 Beta Draft—Cisco Confidential Information

The following modules are not necessary for using the API, but are required to run some sample scripts:

- XML::LibXSLT—This module is needed for the sample scripts that use XSLT to produce HTML pages. The module also requires that the libxslt library be installed on the system first. The toolkit is tested to work with XML::LibXSLT version 1.57 and libxslt version 1.1.3.
- Mail::Send—This module is needed only for the notifications example script.

Installing the Cisco IOS XR Perl Scripting Toolkit

The Cisco IOS XR Perl Scripting Toolkit is distributed in a file named `Cisco-IOS_XR-Perl-Scripting-Toolkit-<version>.tar.gz`.

To install the Cisco IOS XR Perl Scripting Toolkit, perform the following steps:

- Step 1** Extract the contents from the directory in which the file resides by entering the following command:
- ```
tar -f Cisco-IOS_XR-Perl-Scripting-Toolkit-<version>.tar.gz -xzc <destination>
```

Table 15-2 defines the parameters.

**Table 15-2** Toolkit Installation Directory Parameters

| Name          | Description                                                                                                                                                                                                                                        |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <version>     | Defines the version of the toolkit that you want to install, for example, version 1.0.                                                                                                                                                             |
| <destination> | Specifies the existing directory in which you want to create the toolkit installation directory. A directory called Cisco-IOS_XR-Perl-Scripting-Toolkit-<version> is created within the <destination> directory along with the extracted contents. |

- Step 2** Use the **cd** command to change to the toolkit installation directory and enter the following command:
- ```
perl Makefile.PL
```

If the command gives a warning that one of the prerequisite modules is not found, download and install the applicable module from the Comprehensive Perl Archive Network (CPAN) before using the API.

- Step 3** Use the **make** command to maintain a set of programs, as shown in the following example:
- ```
make
```

- Step 4** Use the **make install** command, as shown in the following example:
- ```
make install
```

Ensure that you have the applicable permission requirements for the installation. You may need to have root privileges.

If you do not encounter any errors, the toolkit is installed successfully. The Perl modules are copied into the appropriate directory, and you can use your own Perl scripts.

R3.3 Beta Draft—Cisco Confidential Information

Using the Cisco IOS XR Perl XML API in a Perl Script

To use the Cisco IOS XR Perl XML API in a Perl application, import the module by including the following statement at the top of the script:

```
use Cisco::IOS_XR;
```

If you are using the Data Object interface, you can specify extra import options in the statement. For more information about the objects, see “Creating Data Objects” section on page 15-142.

Handling Types of Errors for the Cisco IOS XR Perl XML API

The following types of errors can occur when using the Cisco IOS XR Perl XML API:

- Errors returned from the router—Specify that the errors are produced during the processing of an XML request and are returned to you in an XML response document. For more information about how these errors are handled, see “Using the Error Objects” section on page 15-137.
- Errors produced within the Perl XML API modules—Specify that the script cannot continue. The module causes the script to be terminated with the appropriate error message. If the script writer wants the script to handle these error types, the writer must write the die handlers (for example, enclose the call to the API function within an eval{ } block).

Starting a Management Session on a Router

Before any requests are sent, a management session must be started on the router, which is done by creating a new object of type named Cisco::IOS_XR. The new object is used for all further requests during the session, and the session is ended when the object is destroyed. A Cisco::IOS_XR object is created by calling Cisco::IOS_XR::new.

Table 15-3 lists the optional parameters specified as arguments.

Table 15-3 Argument Definitions

Name	Description
<i>use_command_line</i>	Controls whether or not the new() method parses the command-line options given when the script was invoked. If the value of the argument is true, which is the default, the command-line options specify or override any of the subsequent arguments and control debug and logging options. The value of 0 defines the value as false.
<i>interactive</i>	<p>If the value of the argument is true, the script prompts you for the username and password if they have not been specified either in the script or on the command line. The Term::ReadKey module must be installed.</p> <p>The most secure way of using the toolkit is for the input not to be echoed to the screen, which avoids hard coding or any record of passwords having been used. The default value is false which means that the script does not ask for user input. As a command-line option, the interactive argument does not take any arguments. You can specify -interactive to turn on the interactive mode.</p>

R3.3 Beta Draft—Cisco Confidential Information**Table 15-3 Argument Definitions (continued)**

Name	Description
<i>transport</i>	Means by which the Perl application should connect to the router, which defaults to Telnet. If a different value is specified, the <code>new()</code> method searches for a package called <code>Cisco::IOS_XR::Transport::<transport_name></code> . If found, the Perl application uses that package to connect to the router.
<i>ssh_version</i>	If the chosen transport option is SSH and the SSH executable on your system supports SSH v2, specifies which version of SSH you want to use for the connection. The valid values are 1 and 2. If the SSH executable supports only version 1, an error is caused by specifying the <i>ssh_version</i> argument.
<i>host</i>	Specifies the name or IP address of the router to which to connect. The router console or auxiliary ports should not be used because they are likely to cause problems for the script when logging in and offer significantly lower performance than a management port.
<i>port</i>	Specifies the TCP port for the connection. The default value depends on the transport being used.
<i>username</i>	Specifies the username to log in to the router.
<i>password</i>	Specifies the corresponding password.
<i>connection_timeout</i>	Specifies the timeout value that is used to connect and log in to the session. If not specified, the default value is 5 seconds.
<i>response_timeout</i>	Specifies the timeout value that is used when waiting for a response to an XML request. If not specified, the default value is 10 seconds.
<i>prompt</i>	Specifies the prompt that is displayed on the router after a successful log in. The default is <code><host>#</code> .

The following example shows the arguments given using the standard Perl hash notation:

```
use Cisco::IOS_XR;
my $session = new Cisco::IOS_XR(transport => 'telnet',
                                host => 'router1',
                                port => 7000,
                                username => 'john',
                                password => 'smith',
                                connection_timeout => 3);
```

Table 15-4 describes the additional command-line options that can be specified.

Table 15-4 Command-Line Options

Name	Description
debug	Turns on the specified debug type and can be repeated to turn on more than one type.
logging	Turns on the specified logging type and can be repeated to turn on more than one type.
log_file	Specifies the name of the log file to use.
telnet_input_log	Specifies the file used for the Telnet input log, if you are using Telnet.
telnet_dump_log	Specifies the file used for the Telnet dump log, if you are using Telnet.

R3.3 Beta Draft—Cisco Confidential Information

To use the command-line options when invoking a script, use the `-option` code formatting (assuming the option has a value). The option name does not need to be given in full, but must be long enough to be distinguished from other options. The following example is displayed:

```
perl my_script.pl -host my_router -user john -interactive -debug xml
```

Closing a Management Session on a Router

When an object of type `Cisco::IOS_XR` is created, the transport connection to the router and any associated resources on the router are maintained until the object is destroyed and automatically cleaned. For most scripts, the process should occur automatically when the script ends.

If you want to close a particular session during the course of the script, use the `close()` method. You can perform an operation on a large set of routers sequentially, and not keep all sessions open for the duration of the script, as displayed in the following example:

```
my $session1 = new Cisco::IOS_XR(host => 'router1', ...);
#do some stuff
$session1->close;
my $session2 = new Cisco::IOS_XR(host => 'router2', ...);
# do some stuff
...
```

Sending an XML Request to the Router

Requests and responses pass between the client and router in the form of XML. Depending on whether the XML is stored in a string or file, you can construct an XML request that is sent to the router using either the `send_req` or `send_req_file` method. Some requests are sent without specifying any XML by using the configuration services methods, for example, `commit` and `lock` or the Data Object interface.

The following example shows how to send an XML request in the form of a string:

```
my $xml_req_string = '<?xml...><Request>...</Request>';
my $response = $session->send_req($xml_req_string);
```

The following example shows how to send a request stored in a file:

```
my $response = $session->send_req_file('request.xml');
```

Using Response Objects

Both of the `send_req` and `send_req_file` methods return a `Cisco::IOS_XR::Response` object, which contains the XML response returned by the router.



Note

Both `send` methods handle iterators in the background; so if a response consists of many parts, the response object returned is the result of merging them back together.

R3.3 Beta Draft—Cisco Confidential Information

Retrieving the Response XML as a String

The following example shows how to use the `to_string` method:

```
$xml_response_string = $response->to_string;
```

Writing the Response XML Directly to a File

The following example shows how to use the `write_file` method by specifying the name of the file to be written:

```
$response->write_file('response.xml');
```

Retrieving the Data Object Model Tree Representation of the Response

The following example shows how to retrieve a Data Object Model (DOM) tree representation for the response:

```
my $document = $response->get_dom_tree;
```

You should be familiar with the DOM, which an XML document represents an object tree structure. For more information, see <http://www.w3.org/DOM/>



Note

The returned DOM tree type will be of type `XML::LibXML::Document`, because this is the form in which the response is held internally. The method is quick, because it does not perform extra parsing and should be used in preference to retrieving the string form of the XML and parsing it again (unless a different DOM library is used).

Determining if an Error Occurred While Processing a Request

The following example shows how to determine if an error occurred while processing a request:

```
my $error = $response->get_error;
if (defined($error)) {
    die $error;
}
```

If it is only of interest whether errors occurred but not what all the errors were, use the `get_error` method to return one error from the response. This returns an error object that represents the first error found or is undefined if none were found.

Retrieving a List of All Errors Found in the Response XML

The following example shows how to list all errors that occurred rather than just one by using the `get_errors` method:

```
my @errors = $response->get_errors;
```

The `get_errors` method returns an array of error objects that represents all errors that were found in the response XML. For more information, see “Using the Error Objects” section on page 15-137.

Using the Error Objects

Error objects are returned when calling the `get_error` and `get_errors` methods on a response object, and are used to represent an error encountered in an XML response. [Table 15-5](#) lists the methods for the object.

R3.3 Beta Draft—Cisco Confidential Information

Table 15-5 List of Methods for the Object

Method	Description
get_message	Returns the error message string that was found in the XML.
get_code	Returns the corresponding error code.
get_element	Returns the tag name of the XML element in which the error was found.
get_dom_node	Returns a reference to the element node in the response DOM ¹ tree.
to_string	<p>Returns a string that contains the error message, code, and element name. If the error object is used in a scalar context, the method is used automatically to convert it to a string. The following example displays all information in an error:</p> <p>Error encountered in object ConfederationPeerASTable: 'XMLMDA' detected the 'warning' condition 'The XML request does not conform to the schema. A child element of the element on which this error appears includes a non-existent naming, filter, or value element. Please check the request against the schema.' Error code: 0x4368a000</p>

1. DOM = Data Object Model

Using the Configuration Services Methods

Methods are provided to enable the standard configuration services operations to be performed without knowledge of the underlying XML. These are the operations that are usually performed at the start or end of a configuration session, such as locking the running configuration or saving the configuration to a file.

The `config_commit()` function takes the following optional arguments:

- *mode*
- *label*
- *comment*

Committing the Target Configuration

The arguments are specified in any order using the standard Perl hash notation, as shown in the following example:

```
$response = $session->config_commit(Label => 'Example1', Comment => 'Just an example');
```

A response object is returned from which any errors can be extracted, if desired. To retrieve the commit ID that was assigned to the commit upon success, you can call the `get_commit_id()` method on the response object, as shown in the following example:

```
$commit_id = $response->get_commit_id();
```

Locking and Unlocking the Running Configuration

The following example shows how to use the `config_lock` and `config_unlock` functions, which take no arguments:

```
$error = $session->config_lock;
$error = $session->config_unlock;
```

R3.3 Beta Draft—Cisco Confidential Information

Loading a Configuration from a File

The following example shows how to contain a filename as an argument:

```
$error = $session->config_load(Filename => 'test_config.cfg');
```

Loading a Failed Configuration

The following example shows how to use the `config_load_failed` function, which takes no arguments:

```
$error = $session->config_load_failed;
```

Saving a Configuration to a File

The following example shows how to use two arguments for the `config_save()` function:

```
$error = $session->config_save(Filename => 'disk0:/my_config.cfg', Overwrite => 'true');
```

The first argument shows how to use the filename to which to write and the Boolean overwrite setting. The filename must be given with a full path. The second argument is optional.

Clearing the Target Configuration

The following example shows how to use the `config_clear` function, which takes no arguments:

```
$error = $session->config_clear;
```

Getting a List of Recent Configuration Commits

The following example shows how to use the `config_get_history()` function that takes the optional arguments *Maximum* and *RollbackOnly*, which can be specified in any order by using the standard Perl hash notation:

```
$response = $session->config_get_history(Maximum => 10, RollbackOnly => 'true');
```

It returns a response object in which the method `get_entries` can be called. This returns an array of entry objects from which the `get_key` method can be called to retrieve the `CommitID` and `get_data` to retrieve the rest of the fields.

Rolling Back to a Previous Configuration

The following example shows how to use the `config_rollback()` function that takes the optional arguments *Label* and *Comment*, and exactly one of the two arguments *CommitID* or *Previous*:

```
$error = $session->config_rollback(Label => 'Rollback test', CommitID => 1000000072);
```

These arguments can be specified in any order using the standard Perl hash notation.

Getting a List of Current Configuration Sessions

The following example shows how to use the `config_get_sessions` function, which takes no arguments:

```
$response = $session->config_get_sessions;
```

It returns a response object in which the method `get_entries` can be called. This returns an array of entry objects in which the method `get_key` method can be called to retrieve the session ID and `get_data` method to retrieve the rest of the fields.

R3.3 Beta Draft—Cisco Confidential Information

Sending a Command-Line Interface Configuration Command

The following example shows how to use the `config_cli()` function, which takes a string argument containing the CLI format configuration that you want to apply to the router:

```
$response = $session->config_cli($cli_command);
```

To retrieve the textual CLI response from the response object returned, you can use the `get_cli_response()` method, as shown in the following example:

```
$response_text = $response->get_cli_response();
```



Note

Apart from the `config_commit`, `config_get_history`, `config_get_sessions`, and `config_cli` methods, each of the other methods returns a reference to an error object if an error occurs or is undefined. For more information, see [“Using the Error Objects” section on page 15-137](#).

Using the Cisco IOS XR Perl Data Object Interface

Instead of having to specify the XML requests explicitly, the interface allows access to management data using a Perl notation. The Data Object interface is a Perl representation of the management data hierarchy stored on the router. It consists of objects of type `Cisco::IOS_XR::Data`, which corresponds to items in the IOS_XR management data hierarchy, and a set of methods for performing data operations on them.

To use the Data Object interface, a knowledge is needed of the underlying management data hierarchy. The management data on an IOS_XR router all under one of six `root` objects, namely `Configuration`, `Operational`, `Action`, `AdminConfiguration`, `AdminOperational`, and `AdminAction`. The objects that lie below these objects in the hierarchy, along with definitions of any datatypes or filters that are used by them, are documented in the Perl Data Object Documentation.

A hash structure is defined to be a scalar (that is, basic) type, for example, string or number, a reference to a hash whose values are hash structures, or a reference to an array whose values are hash structures. This standard Perl data structure corresponds naturally to the structure of management data on an IOS_XR router. The following example shows how to use a hash structure:

```
# basic type
my $struct1 = 'john';
# reference to a hash of basic types
my $struct2 = {Forename => $struct1, Surname => 'smith'};
# reference to an array of basic types
my $struct3 = ('dog', 'budgie', 'cat');
# reference to a hash of references and basic types
my $struct4 = {Name => $struct2, Age => '30', Pets => $struct3};
```

The following sections describe how to use the Perl Data Object Documentation:

- [Understanding the Perl Data Object Documentation, page 15-141](#)
- [Generating the Perl Data Object Documentation, page 15-141](#)
- [Creating Data Objects, page 15-142](#)
- [Specifying the Schema Version to Use When Creating a Data Object, page 15-143](#)
- [Using the Data Operation Methods on a Data Object, page 15-144](#)
- [Using the Batching API, page 15-147](#)
- [Displaying Data and Keys Returned by the Data Operation Methods, page 15-148](#)

R3.3 Beta Draft—Cisco Confidential Information

- [Specifying the Session to Use for the Data Operation Methods, page 15-149](#)

Understanding the Perl Data Object Documentation

The Perl Data Object Documentation consists of many files, each containing a subtree of the total management data hierarchy. The main part of each filename tells you the area of management data to which that file refers, and the suffix usually tells you which root object that file's data lies below. For example, a file containing configuration data usually ends in `_cfg.html`. There may also be some files that do not contain any object definitions but just some datatypes or filter definitions, and will usually end in `_common.html`.

For leaf objects, the object definition describes the data that the object contains. For nonleaf objects, the definition provides a list of the object's children within the tree. More precisely, the object definition consists of the following items:

- Name of the object.
- Brief description of what data is contained in the object or in the subtree below.
- List of the required task IDs that are required to access the data in the object and subtree.
- List of parent objects and the files in which they are defined, if the object is the top-level object in that file.
- If the object is a leaf object (for example, data is contained without child objects), and its name is not unique within that file, parent objects are listed.
- If the object is a table entry, a list of the keys that are needed to identify a particular item in that table. For each key, a name, description, and datatype are given.
- If the object is a table, a list of the filters that can be applied to that table.
- If the object is a leaf object, a list of the value items that are contained. For each value item, a name, description, and datatype are given.
- If the object is a leaf object, its default value (for example, the values for each of its value items that would be assumed if the object did not exist), if there is one.
- List of the data operation methods, `get_data`, `set_data`, and so forth that are applicable to the object. For more information, see [“Specifying the Schema Version to Use When Creating a Data Object” section on page 15-143](#)

Generating the Perl Data Object Documentation

The Perl Data Object Documentation must be generated from the schema distribution tar file “All-schemas-CRS-1-”release”.tar.gz”, where “release” is the release of the CRS software that you have installed on the router.

From the `perl` subdirectory under the extracted contents of the above-mentioned Schema tarball, copy all `*.dat` files into the toolkit installation directory `Cisco-IOS_XR-Perl-Scripting-Toolkit-”version”/dat`. (default) or a selected directory for the `.dat` files. These `.dat` files are the XML files that will be used to generate the HTML documentation.

From the `perl` subdirectory under the extracted contents of the above-mentioned Schema tarball, copy all the `*.html` files into the toolkit installation directory `Cisco-IOS_XR-Perl-Scripting-Toolkit-”version”/html` (default) or a selected directory for the `.html`.

R3.3 Beta Draft—Cisco Confidential Information

(The default .html subdirectory already contains two html files that were extracted with the Toolkit distribution - root_objects.html and common_datatypes.html . These are automatically copied to the selected .html directory, if a non-default directory is selected, upon performing the following step).

Run the script generate_html_documentation.pl which is available in the distribution Cisco-IOS_XR-Perl-Scripting-Toolkit-"version"/scripts directory, giving the appropriate directories for the .dat and .html files, when prompted.

If the script fails indicating any error .dat files, evaluate the .dat file to confirm that it is not of "0" size and that it has a header like the following :

```
<?xml version="1.0" encoding="UTF-8"?>

<!--

Copyright (c) 2004-2005 by cisco Systems, Inc.
All rights reserved.
```

If not, then remove the .dat file and re-run the script.

Linked HTML files will be created in the selected (or default) html dirextory. The Perl Data Object API documentation can be traversed using the links starting at root_objects.html.

Creating Data Objects

Data objects form a tree corresponding to a section of the data hierarchy. The first object to be created is one of the root data objects, and is created by a call to Cisco::IOS_XR::Data::<object_name>. For example, <object_name> is one of the following objects:

- Configuration
- Operational
- Action
- AdminOperational
- AdminAction

The following example shows how to create the Operational object:

```
my $oper = Cisco::IOS_XR::Data::Operational;
```

Because the syntax is rather lengthy for a task that is relatively common, there is a shorter way of creating a data object, which eliminates the need for the Cisco::IOS_XR::Data:: at the front of the function name. This is achieved by importing the symbols for the root data object functions when using the Cisco::IOS_XR package at the top of the script. The following example shows how to import the Configuration and Operational functions:

```
use Cisco::IOS_XR qw(Configuration Operational);
```

The following example shows how to import all the root data objects without listing them explicitly:

```
use Cisco::IOS_XR qw(:root_objects);
```



Note

If there is a function in the script's name space with a name that is one of Configuration, Operational, and so forth, the root data objects cannot be imported with use Cisco::IOS_XR qw(Configuration Operational) and refer to the objects simply as Configuration, as this may not have the desired effect due to the ambiguity. Instead, you have to refer to them with the more lengthy Cisco::IOS_XR::Data::Configuration (that is, fully qualified) syntax.

R3.3 Beta Draft—Cisco Confidential Information

If the root data object is Configuration, additional arguments can be specified that are given as name and value pairs. The *Source* argument can have values such as ChangedConfig, CurrentConfig, MergedConfig (the default value if the *Source* argument is not specified), and CommitChanges. If CommitChanges is specified, one of the two arguments *ForCommitID* and *SinceCommitID* must also be specified, as shown in the following example:

```
my $config = Configuration(Source => 'CommitChanges', ForCommitID => 1000083);
```

Data objects can be created from existing ones by calling a method on the existing object for which the name is that of the new object that you want to create. The object from which the new object was created is known as its parent, as shown in the following example:

```
my $config = Configuration;
my $bgp = $config->BGP;
```

If references to the intermediate objects are not required, the syntax allows a very compact way of creating objects as the methods can be strung together. The following example shows how to create a BGP object whose parent is Configuration:

```
my $bgp = Configuration->BGP;
```

If an object is an item in a table, its keys can be specified as arguments when the object is created by using the standard Perl hash notation. The following example shows how to create an object corresponding to the interface configuration for interface Ethernet 0/0/0/0:

```
my $if_conf = Configuration->InterfaceConfigurationTable->
    InterfaceConfiguration('Active' => 'act', 'Name' => 'Ethernet0/0/0/0');
```

Keys can also be specified by passing a hash structure as an argument. The hash structure would usually have been returned as a key from one of the data operation methods, for example, `get_keys`, but can be defined explicitly, as in the following alternative to the previous example:

```
my $key = {'Active' => 'act', 'Name' => 'Ethernet0/0/0/0'};
my $if_conf = Configuration->InterfaceConfigurationTable->InterfaceConfiguration($key);
```

There may be some occasions when it is better to keep references to the intermediate data objects, such as when you want to refer to more than one item in a table. The following example shows how to refer to more than one interface in the interface configuration table:

```
my $if_1_key = {'Active' => 'act', 'Name' => 'Ethernet0/0/0/0'};
my $if_2_key = {'Active' => 'act', 'Name' => 'POS0/4/0/0'};
my $if_conf_table = Configuration->InterfaceConfigurationTable;
my $if_conf_1 = $if_conf_table->InterfaceConfiguration($if_1_key);
my $if_conf_2 = $if_conf_table->InterfaceConfiguration($if_2_key);
```

Currently, there is no checking within the library that the object names specified are valid. However, when a data operation is performed on a data object, and if the object hierarchy is invalid, the response from the router should contain an error to this effect. For information on the valid object names in the data hierarchy, see [“Understanding the Perl Data Object Documentation”](#) section on page 15-141.

Specifying the Schema Version to Use When Creating a Data Object

If you want to specify which version of a particular schema you are using, you may pass this information as arguments when creating the relevant data object. The router checks this information against its own schema versions when it receives a request, and rejects the request if the versions are not compatible. For more information about versioning, see [Chapter 9, “Cisco XML Schema Versioning.”](#)

R3.3 Beta Draft—Cisco Confidential Information

The object in which this information should be specified is the top-level object within the schema whose version you want to specify. This information is found at the top of the page of the schema. For more information, see [“Understanding the Perl Data Object Documentation” section on page 15-141](#). For example, you may want to specify that using BGP schema version 1.4. The following example shows how to create a BGP object:

```
my $bgp = Configuration->BGP(MajorVersion => 1, MinorVersion => 4);
```

The object can then be used in the normal way to create child objects. Whenever any data operation request is sent using one of these objects, the specified version information is always included.

Using the Data Operation Methods on a Data Object

To access the management data on the router, data operation methods, which can be called on data objects, are provided for the getting, setting, and deletion of the corresponding data. The management session in which they act is the current session, and usually the most recent Cisco::IOS_XR object to be created. For more information on how to manually set which session to use for the data operation methods, see [“Specifying the Session to Use for the Data Operation Methods” section on page 15-149](#).

The types of data operation methods that are allowed depend on what the root data object is for the data object in question. For example, if the root object is Configuration, getting, setting, and deletion are allowed. If it is Operational, only getting is allowed. The get methods that can be used also depend on whether the data object in question is a leaf object or a table object.

Each of the data operation methods returns a response object from which any errors can be extracted. For more information, see [“Using Response Objects” section on page 15-136](#). For the methods that return values of some sort, a method of the same name is used to actually extract the information required from the response object.

R3.3 Beta Draft—Cisco Confidential Information

get_data Method

The `get_data` method can be called on a leaf object and is used to retrieve the data contained in that object. It returns a response object from which the desired data can be extracted by calling the method of the same name, `get_data`.

The following example shows how to get the data for the interface configuration:

```
my $response = $if_conf->get_data;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    my $data = $response->get_data;
    ...
}
```

find_data Method

The `find_data` method performs a get request on a leaf object, but with the option of specifying key values for any table entries that occur within the hierarchy as a wildcard rather than as explicit values.

The XML response then contains every occurrence of the required object that matches the combination of key values and wildcards specified in the hierarchy.



Note

Wildcards are supported for only configuration data.

Currently, the function does not interpret the XML response in any way, due to the potentially complex structure of the returned data, and so the returned response object can be used only to extract the XML and any other errors in the usual way.

When specifying the keys for a table entry object, if you want one of the keys to be a wildcard rather than specified explicitly, pass an argument called `wildcard` value, where the value is the name of the key. If access control lists (ACLs) have been configured, the following example shows how to get the inbound ACLs of all interfaces on the router:

```
my $response = $if_conf_table->
    InterfaceConfiguration(Active => 'act', wildcard => 'Name')->
    IPV4PacketFilter->Inbound->find_data;
```

If you want one or more of the keys for a particular table entry to be wildcards, the value of the wildcard can be a reference to an array containing the names of those keys. For example, if you want to include any nonactive interface configuration in the above example, you would do the following:

```
my $response = $if_conf_table->
    InterfaceConfiguration(wildcard => ['Name', 'Active'])->
    IPV4PacketFilter->Inbound->find_data;
```

get_keys Method

The `get_keys` method must be called only on a table object and is used to retrieve a list of the keys for each item in the table. It returns a response object from which the keys can be extracted by calling the method of the same name, `get_keys`. This returns an array of hash structures containing the key values. A returned key can also be used as the parameter to a new data object.

R3.3 Beta Draft—Cisco Confidential Information

The following example shows how to get the keys for each item in the configuration table and then for each key to create a data object and perform some operations with it:

```
my $response = $if_conf_table->get_keys;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    foreach my $key ($response->get_keys) {
        my $interface = $if_conf_table->InterfaceConfiguration($key);
        # do something with this object such as get_data...
    }
}
```

The following two optional arguments can be specified as name and value pairs:

- **Count**—Determines the maximum number of table entries that will be returned.
- **Filter**—Specifies a reference to a hash whose elements are the arguments to the filter plus an element `Filtername` that specifies the filter to use, as shown in the following example:

```
my $table = Operational->BGP->VRFTTable->VRF(VrfName='VRF1')->NeighborTable;
my $filter = {FilterName => 'BGP_ASFilter', AS => 6};
my $response = $table->get_keys(Count => 10, Filter => $filter);
```

get_entries Method

Similarly, the `get_entries` method must be called only on a table object, and is used to retrieve a list of the keys and data for each entry in the table.

It returns a response object from which the entries can be extracted by calling the method of the same name, `get_entries`. This method returns an array of entry objects. The `get_key` and `get_data` methods can then be called on an entry object to extract the key and data for that entry.

The following example shows how to get an array of the keys and data for each item in the interface configuration table and perform some operations with each:

```
my $response = $if_conf_table->get_entries;
if (defined($response->get_error)) {
    die $response->get_error;
} else {
    foreach my $entry ($response->get_entries) {
        my $key = $entry->get_key;
        my $data = $entry->get_data;
        # do something with these values...
    }
}
```

The same optional arguments, `Count` and `Filter`, can be specified in the `get_keys` method.

set_data Method

The `set_data` method is called only on leaf objects, and sets the data for the object in the specified argument. The argument must be a hash structure; for example, the data is returned by a previous call of `get_data` or `get_entries`.

The returned value is a response object from which the entries are extracted. Unless batching is enabled, the returned value is undefined.

R3.3 Beta Draft—Cisco Confidential Information

The following example shows how to add a IPv4Multicast object to the GlobalAFTable of BGP AS 1 object:

```
my $data = {'Enabled' => 'true'};
my $global_af = Configuration->BGP->AS('AS' => 1)->DefaultVRF->Global->
    GlobalAFTable->GlobalAF('AF' => 'IPv4Multicast');
my $error = $global_af->set_data($data);
```



Note

- If not all items in a leaf object are specified when setting data, the remaining items are set to null (overwrites any value that may have been there previously).
- If the data is passed to set_data as a hash or basic type (not an array), it can also be provided explicitly rather than by reference, in the same way as keys can be specified.

The following example shows how data that is passed to set_data as a hash or basic type:

```
my $error = $global_af->set_data('Enabled' => 'true');
```

If the data to be set is an array, it must be provided by references because if it were given explicitly it would be incorrectly interpreted as a hash.

delete_data Method

The delete_data method can be used on any object, and deletes all data below that object in the hierarchy, as shown in the following example:

```
my $error = Configuration->BGP->AS('AS' => 1)->DefaultVRF->Global->delete_data;
```

The returned value is a response object from which any errors can be extracted. Unless batching is enabled, the returned value is undefined.

Using the Batching API

By default, whenever the set_data or delete_data methods are called on a data object, the resulting XML request is sent immediately. The script is enabled to verify immediately whether or not the operation was successful. However, if a script wants to set or delete many items at once, this can be a very inefficient method.

By using the batching API, a script can specify that it wants a group of set or delete operations all to be sent together in one XML request. This reduces the overhead of the router having to process multiple requests and reduces the amount of data that needs to be sent. Due to the way two XML requests with overlapping hierarchies are merged, the resulting XML is not as long as the sum of the original two. The common hierarchy is not repeated.



Note

A commit operation cannot be performed within a batch. To enforce this, the config_commit() function dies with an error if it is called while batching is in progress.

R3.3 Beta Draft—Cisco Confidential Information

batch_start Method

When the `batch_start` method is called on the session object in question, all subsequent calls of `set_data` or `delete_data` are not performed immediately but are stored locally until the `batch_send` method is called. The following example shows how to enable batching on the session `$session`:

```
$session->batch_start;
```

**Note**

Any calls to `set_data` or `delete_data` between the `batch_start` and the subsequent `batch_send` methods return undefined rather than as a response object

batch_send Method

The `batch_send` method should be called at the point in the script when you want to send all set and delete operations that were made since the previous call to `batch_start`. The `batch_send` method sends these operations as a single XML request and returns a single response object. If this response contains no errors, all operations were successful. Otherwise, the details of any error returned must be analyzed to determine which operation caused the error, as shown in the following example:

```
my $response = $session->batch_send;
my $error = $response->get_error;
if ($error) {
    die "Error in batch_send: $error";
}
```

**Note**

An error occurs in the script if `batch_send` is called while batching is not in progress, for example, it must occur after a call to `batch_start`.

Displaying Data and Keys Returned by the Data Operation Methods

When a key or data is returned either by calling `get_data` or `get_keys` functions on a response object, or by calling `get_data` or `get_key` functions on an entry object that was returned from the `get_entries` function, it is always in the form of a value object. This object behaves identically to a hash structure; therefore, the value object can be easily navigated using hash and array dereferencing if required. A key value can be used when creating a new data object or as an argument to the `set_data` function.

However, if you want to display the whole structure or any parts of it, use the built-in function `to_string` on any value object that returns a formatted string form of the structure. In fact, you do not need to call the function `to_string` on the object. Using the value object in a scalar context, automatically converts it to a formatted string. The following code is shown:

```
my $response = Configuration->InterfaceConfigurationTable
    ->InterfaceConfiguration(Active => 'act', Name => 'POS0/2/0/0')
    ->get_data;
print $response->get_data;
```

R3.3 Beta Draft—Cisco Confidential Information

The following example displays that data on the screen in a readable way:

```
Shutdown
  true
IPV4PacketFilter
  Inbound
    HardwareCount
    Name
      myacl
Description
  my POS interface
```

Specifying the Session to Use for the Data Operation Methods

If only one Cisco::IOS_XR object has been created, this management session is automatically used by subsequent data operation methods. In scripts in which more than one Cisco::IOS_XR object has been created, the data operation methods use whichever session is the current session. The session to use for the data operation methods is whichever Cisco::IOS_XR object was the last to be created, unless, you have since asked to change the current session by calling the method `use_for_data_operations` on the Cisco::IOS_XR object that you want to use.

The following example shows how to create two management sessions and then use the first one for subsequent data operations:

```
my $session1 = new Cisco::IOS_XR(host => 'router1');
# Here the current session is $session1
my $session2 = new Cisco::IOS_XR(host => 'router2');
# Here the current session is $session2
$session1->use_for_data_operations;
# Now the current session is $session1 again
```

Cisco IOS XR Perl Notification and Alarm API

The notification API provides functionality that enables a Perl script to register for and receive asynchronous responses or notifications during a management session on the router. One important type of notification is Alarms for which the specific API is provided.

The API allows a script to register, deregister, and receive alarms using Perl methods and objects. This completely hides the underlying XML from the user in much the same way that the data object API for normal management requests does.

The following sections describe how to use the Alarm API:

- [Registering for Alarms, page 15-150](#)
- [Deregistering an Existing Alarm Registration, page 15-150](#)
- [Deregistering All Registration on a Particular Session, page 15-150](#)
- [Receiving an Alarm on a Management Session, page 15-150](#)

R3.3 Beta Draft—Cisco Confidential Information

Registering for Alarms

To register for a receipt of alarms on a particular management session, use the `alarm_register` function of the `Cisco::IOS_XR` object that represents the management session. The `alarm_register` function takes as arguments a list of name and value pairs, which specify the set of filter criteria that you want to use to filter the alarms that you receive. If no filter criteria are specified, all alarms are received. For a list of the valid filter criteria, see the “Alarm Registration” section on page 10-107 of Chapter 10, “Alarms.”

The following example shows how to register for receipt of all alarms of Group SYS and Code CONFIG_I:

```
my $response = $session->alarm_register(Group => 'SYS', Code => 'CONFIG_I');
```

The `alarm_register` function returns a response object that is checked for errors in a normal way. These errors may be returned if a value specified for one of the filter criteria is invalid.

In addition, a successful registration response contains a registration ID, which must be used if the script wants to deregister. In other words, cancel this registration. The registration ID can be extracted from the response object by calling the `get_registration_id` method, as shown in the following example:

```
my $registration_id = $response->get_registration_id;
```

Deregistering an Existing Alarm Registration

To deregister a particular registration, use the `alarm_deregister` function on the `Cisco::IOS_XR` object, by giving as an argument for the registration ID that was returned from the initial registration as follows:

```
my $response = $session->alarm_deregister($registration_id);
```

The response object that is returned is checked for errors to determine if the deregistration was successful.

Deregistering All Registration on a Particular Session

To deregister all alarm registrations that have been made on a particular management session, use the `alarm_deregister_all` function as follows:

```
my $response = $session->alarm_deregister_all;
```

The response object can be used to check for any errors. Currently, no errors should exist even if there was no registration to deregister on that session.

Receiving an Alarm on a Management Session

After alarms have been registered, the `alarm_receive` function can be called on the management session object. The `alarm_receive` function attempts to pick up an alarm from the transport, which may happen immediately if there is already an alarm waiting in the buffer. Otherwise, it waits until one is received. An optional timeout value can be specified as the argument. If an alarm is not received within the timeout limit, the function returns undefined. If no timeout value is specified, the default of an infinite timeout is used, as shown in the following example:

```
my $alarm = $session->alarm_receive(60); # Wait 60 seconds for an alarm
```

R3.3 Beta Draft—Cisco Confidential Information

If an alarm is received within the timeout limit, the function returns an alarm object from which the following values can be extracted:

- RegistrationID—Specifies the registration ID that was returned from the registration for the matched alarm.
- SourceID
- EventID
- Timestamp
- Category
- Group
- Code
- Severity
- State
- CorrelationID
- AdditionalText

These values can be extracted using the corresponding `get_*` functions, as shown in the following example:

```
my $registration_id = $alarm->get_registration_id;  
my $event_id = $alarm->get_event_id;  
my $text = $alarm->get_additional_text;
```

Using the Debug and Logging Facilities

The following sections describe how to control debug and logging facilities within your script:

- [Debug Facility Overview, page 15-151](#)
- [Logging Facility Overview, page 15-152](#)

For more information on how to control debug and logging from the command line when starting a script, see [“Starting a Management Session on a Router” section on page 15-134](#).

Debug Facility Overview

The debug facility displays on the screen run-time information to aid investigation of problems. The user is given fine control over which debug messages are displayed to the screen by allowing the user to specify at any point in the script which types of debug they want to be displayed and which ones they do not.

**Note**

Debug applies to the script as a whole rather than to each management session.

R3.3 Beta Draft—Cisco Confidential Information

Table 15-6 lists the current built-in types.

Table 15-6 Definitions for the Debug Types

Type	Description
transport	Specifies the messages relating to the state of the current transport, for example, Telnet or SSH.
xml	Displays the request and response XML for every request sent to the router that includes those generated by the Data Object interface and configuration services methods.
xml_response_parts	Displays each part separately if an XML response has been split into multiple parts.
user	Specifies that the script writer can be used to add his or her own debug messages.

To turn on debug, use the `Cisco::IOS_XR::debug_on` function at any point in your script, giving those types of debug that you want to turn on as arguments. This is shown in the following example:

```
Cisco::IOS_XR::debug_on('transport', 'xml');
```

Similarly, to turn off debug for certain types, use the `Cisco::IOS_XR::debug_off` function. Specifying no arguments turns off all types of debug, as shown in the following example:

```
Cisco::IOS_XR::debug_off('xml');
```

To insert your own debug messages in a script, use the `Cisco::IOS_XR::debug` function, giving as arguments the type of debug followed by the message. This is shown in the following example:

```
Cisco::IOS_XR::debug('user', 'This is a user debug message');
```

In addition to being able to use the built-in type `user` to add debug messages to the scripts, it is possible to define your own debug types to give greater control over what is displayed. This is done by calling the `Cisco::IOS_XR::add_debug_types` function and giving as arguments a list of name and value pairs. The name is the name of the new type, and the value is its display name (that is, the string that appears at the beginning of every message of that type when displayed on the screen at run time). This is shown in the following example:

```
Cisco::IOS_XR::add_debug_types('general' => 'General', 'detailed' => 'Detailed');
```

These types can immediately be used to write debug messages, as shown in the following example:

```
Cisco::IOS_XR::debug('detailed', 'This is a detailed debug message');
```

Logging Facility Overview

The logging facility leaves an audit trail of usage or diagnoses problems after an error has occurred. The types of logging messages that are supported include all debug types, including any user-defined debug types.

To turn on logging, use the `Cisco::IOS_XR::logging_on` function at any point in your script, giving those types of messages that you want to turn on for logging as arguments. This is shown in the following example:

```
Cisco::IOS_XR::logging_on('transport', 'xml');
```


R3.3 Beta Draft—Cisco Confidential Information

Similarly, to turn off logging for certain types, use the `Cisco::IOS_XR::logging_off` function. Logging can be turned off for all types of messages by giving no arguments, as shown in the following example:

```
Cisco::IOS_XR::logging_off('xml');
```

By default, the messages will be written to a file called `ios_xr_log.txt` in the same directory as the running script. You can specify which file to use with the function `Cisco::IOS_XR::set_log_file` that can be called at any point in your script. For example, you may want to specify a different log file before carrying out operations on a different management session, as shown in the following example:

```
Cisco::IOS_XR::set_log_file('router2_log.txt');
```

In addition to being able to log each of the standard message types, the Telnet module allows two types of extra logging at a lower level. These can be turned on for the duration of a management session by specifying one of the following arguments when calling `Cisco::IOS_XR::new`, as listed in [Table 15-7](#).

Table 15-7 Logging Arguments

Type	Description
<code>telnet_input_log</code>	Logs all data received from the router, which usually includes the echoes of everything that is sent.
<code>telnet_dump_log</code>	Logs all I/O ¹ through the Telnet connection in a dump format. The dump, however, is less readable than the input log.

1. I/O = input/output

The value of each argument specifies the file to which the log should be written.



Note

If both types of logging are specified, the filenames must be different and they both must be different from the name of the standard log file.

Examples of Using the Cisco IOS XR Perl XML API

The following sections provide examples of using the Cisco IOS XR Perl XML API to perform some of the common router management tasks:

- [Configuration Examples, page 15-154](#)
- [Operational Examples, page 15-161](#)

The examples demonstrate the advantages of using the XML and Perl XML API instead of the CLI and existing screen-scraping techniques.

They are also intended to show how simple it is to convert the most common configuration and operational tasks to scripts using the Perl XML API, as well as how easy it is to write scripts to perform tasks that are not possible using the existing methods.

Some of these tasks may be quite involved, so sample scripts have been provided within the toolkit, which can be customized to suit your needs. Other tasks may require very few lines of code.

Those examples in which scripts have been provided have a line at the top of the script, which specifies the perl executable to use to run it. By default, this line is `#!/usr/bin/perl -w`. If this is not the location of perl on your machine, you must change this line accordingly before being able to run the script.

R3.3 Beta Draft—Cisco Confidential Information

You may also need to give yourself execute permission on the script if it is not already set using the following **chmod** command:

```
chmod +x <script name>.pl
```

You should be able to run the script using the following command from the directory in which it resides:

```
./<script name>.pl
```

Configuration Examples

Examples are provided for setting the configuration and getting the running configuration, which are two of the most common configuration tasks. Additional examples cover the standard router applications. One of these examples also demonstrates in detail how you would use the Data Object documentation to help write the necessary code to access a particular item of data.

Another example shows how to use the Cisco IOS XR Perl Notification API to perform actions whenever particular events occur, such as getting the current configuration changes whenever a commit occurs, or sending an e-mail to notify an administrator when an interface is down.



Note

- In all basic examples of setting a configuration, the final step of committing the configuration is omitted to avoid repetition.
- All the examples are written as though the script begins with a `use` statement, which imports all root data object functions, such as Configuration, Operational, and Action, as shown in the following example:

```
use Cisco::IOS_XR qw(:root_objects);
```

If your script cannot import the functions due to name clashes, you must fully qualify the function names with the `Cisco::IOS_XR::Data::` prefix.

Setting the IP Address of an Interface

Setting the IP address of an interface is normally performed by a sequence of two CLI commands, as shown in the following example:

```
interface MgmtEth0/0/CPU0/0
ip address 1.2.3.4 255.255.255.0
```

To carry out this example in a Perl script using the Perl Data Object API requires only one line of code, although in practice you would usually break it up into smaller lines for clarity and to be able to reuse parts of it. This is shown in the following example:

```
my $config = Configuration;
my $if_conf_table = $config->InterfaceConfigurationTable;
my $eth0 = $if_conf_table->
    InterfaceConfiguration(Active => 'act', Name => 'MgmtEth0/0/CPU0/0');
$eth0->IPv4Network->Addresses->Primary->
    set_data(IPAddress => '1.2.3.4', Mask => '255.255.255.0');
```

R3.3 Beta Draft—Cisco Confidential Information

If the script is needed to access some other configuration, it would not need to repeat the first line but could use `$config`. Similarly, if it is needed to access some other configuration associated with `Ethernet0/0/0/0`, it would not need to repeat the first three lines but could just use the `$eth0` variable. The following example shows how to set the MTU of the `Ethernet0/0/0/0` to 1500 interface:

```
$eth0->IPV4Network->MTU->set_data(1500);
```



Note

The code as shown in given examples would probably be used in the middle of a large script, which performs a more complex job. If it is a common task, it could also be wrapped in a small function for that purpose.

For example, a function to set up the IP address of an interface could be made very easily by using the preceding code, and is then called, as shown in the following example:

```
set_int_ip_address('Ethernet0/0/0/0', '1.2.3.4', '255.255.255.0');
```

The code could be wrapped in a small script that enabled the task to be performed from the command line, as shown in the following example:

```
set_int_ip_address.pl -name Ethernet0/0/0/0 -ip 1.2.3.4 -mask 255.255.255.0
```

Configuring a Simple BGP Neighbor

The following example shows a correspondence to the CLI commands and subcommands for configuring a Border Gateway Protocol (BGP) neighbor:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router bgp 1
RP/0/RP0/CPU0:router(config-bgp)# neighbor 1.2.3.4
```

The equivalence of these two commands using the Data Object interface is shown in the following example:

```
my $bgp_entity = Configuration->BGP->AS(AS => 1)->BGPEntity;
my $neighbor = $bgp_entity->NeighborTable->
    Neighbor(IPAddress => {IPV4Address => '1.2.3.4'});
```

The following example shows how to set the remote autonomous system (AS) number for the neighbor:

```
$error = $neighbor->RemoteAS->set_data(44);
```

You may want to set the description for this neighbor, as shown in the following example:

```
$error = $neighbor->Description->set_data('The router next door');
```

Adding a List of Neighbors to a BGP Neighbor Group

This is a more complex example, which shows how a script can be used to expedite a common task. You may want perform this task using the CLI. You would have to enter a series of commands, as shown in the following example:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router bgp 1
RP/0/RP0/CPU0:router(config-bgp)# neighbor 1.2.3.4
RP/0/RP0/CPU0:router(config-bgp-nbr)# use neighbor-group user1
RP/0/RP0/CPU0:router(config-bgp-nbr)# exit
RP/0/RP0/CPU0:router(config-bgp)# neighbor 2.3.4.5
```

R3.3 Beta Draft—Cisco Confidential Information

```
RP/0/RP0/CPU0:router(config-bgp-nbr)# use neighbor-group user1
RP/0/RP0/CPU0:router(config-bgp-nbr)# exit
RP/0/RP0/CPU0:router(config-bgp)# neighbor 3.4.5.6
RP/0/RP0/CPU0:router(config-bgp-nbr) use neighbor-group user1

etc...
```

The sample shows how to perform this task in a faster and more user-friendly way, as shown in the following example:

```
./add_neighbors_to_group.pl -host my_router -user john
Password:
Neighbor-group: user1
Neighbor ip addresses to add:
1.2.3.4
2.3.4.5
3.4.5.6
<cr>
```

The script can be found in the examples/bgp/add_neighbors_to_group.pl file within the toolkit installation directory.

Displaying the Members of Each BGP Neighbor Group

The example shows how a script using the Cisco IOS XR Perl scripting toolkit can retrieve and display information in ways that cannot be done using the CLI on the router. The script allows you to display the current members of each neighbor group and, which groups oppose how the information can be viewed using the CLI. In the same way, the previous example shows you how to add neighbors to a group rather than to add the group to each neighbor.

The script can be found in the examples/bgp/display_neighbor_group_members.pl file.

Setting Up ISIS on an Interface

The simplest integrated Intermediate System-to-Intermediate system (ISIS) configuration task is to set up ISIS on an interface. The following CLI commands example is set up as though the interface in question is already configured:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router isis 1
RP/0/RP0/CPU0:router(config-isis)# net 49.0000.0000.3.00
RP/0/RP0/CPU0:router(config-isis)# interface POS0/2/0/0
RP/0/RP0/CPU0:router(config-isis-if)# address-family ipv4
```

To use the Data Object interface, define the ISIS instance. This is shown in the following example:

```
my $instance = Configuration->ISIS->InstanceTable->Instance(InstanceID => 1);
```

The following example shows how to set up a Network Entity Title (NET) for that instance:

```
$instance->NETTable->NET(NET => '49.0000.0000.3.00')->set_data('True');
```

The following example shows how to set up the interface:

```
my $if = $instance->InterfaceTable->Interface(Name => 'POS0/2/0/0');
$if->Running->set_data('True');
```

R3.3 Beta Draft—Cisco Confidential Information

The following example shows how to set up the IPv4 address family on that interface:

```
$if->InterfaceAddressFamilyTable->
  InterfaceAddressFamily(AF => 'IPv4', SubAF => 'Unicast')->
  Running->set_data('True');
```

Finding the Circuit Type That is Currently Configured for an Interface for ISIS

The example shows how to use the Perl Data Object documentation to help you write code to access a particular piece of data.

You may know that ISIS is configured on a particular interface, for example, POS 0/2/0/0, but you may not know whether that interface was configured as only Level 1, only Level 2, or both. The data you are looking for is ISIS configuration data, which should be documented in a file whose name ends with `_cfg.html`. A quick browse through the Data Object documentation files reveals `isis_cfg.html` as a sensible place to look.

The first object definition is ISIS. The parent object is specified as `RootCfg`, which is the top-level configuration object that is accessed using the `Configuration` function. The following example shows how to create an object that corresponds to ISIS configuration:

```
my $isis = Configuration->ISIS;
```

You found that the only child object of ISIS is `InstanceTable` which has entries of the object instance. Under the `Keys` heading, you notice that `Instance` has only one key called `InstanceID`. If the ISIS instance that you are interested in is 1, you can now create a data object corresponding to that instance by specifying the instance ID as an argument. This is shown in the following example:

```
my $instance = $isis->InstanceTable->Instance(InstanceID => '1');
```

Browsing at the child objects of `Instance`, you see an object called `InterfaceTable`, and you want this item of data for a particular interface. Therefore, it is presumably somewhere under that object, as shown in the following example:

```
my $interface_table = $instance->InterfaceTable;
```

By looking at the definition of the `InterfaceTable` object, you see it has one child called `Interface`. Looking at the definition of `Interface`, you see that it has one child called `CircuitType`, which must be the item that you are looking for. The definition of `Interface` contains one key called `InterfaceName`, which specifies the interface to create the corresponding data object, as shown in the following example:

```
my $interface = $interface_table->Interface(InterfaceName => 'POS0/2/0/0');
```

The following example shows how to create a `CircuitType` object:

```
my $circuit_type = $interface->CircuitType;
```

Looking at the definition of `CircuitType`, you see that it has a value and the `get_data` method can be called on it. You can now retrieve the required data, as shown in the following example:

```
my $response = $circuit_type->get_data;
```

The actual value can now be accessed from the response object, as shown in the following example:

```
my $value = $response->get_data;
```

R3.3 Beta Draft—Cisco Confidential Information

You may not want to perform so many steps, and it is probably not necessary. In practice, you may do some of the steps at once. The following sample code would do the same thing:

```
my $response = Configuration->ISIS->InstanceTable->Instance(InstanceID => '1')
    ->InterfaceTable->Interface(InterfaceName => 'POS0/2/0/0')
    ->CircuitType->get_data;
my $value = $response->get_data;
```

Finally, you would have to know the type of value of CircuitType to do any comparison of the value, which is given as ISISConfigurableLevels. The definition of ISISConfigurableLevels states what values are valid for this item. The following enumerations are included with the valid values:

- Level1
- Level2
- Levels1And2

Configuring a New Instance, Area, and Interface for OSPF

The example shows how to set up OSPF on an interface.

Assuming that the POS0/2/0/0 and Loopback0 interfaces are already configured, the following example shows how to use the CLI commands:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# router ospf 1
RP/0/RP0/CPU0:router(config-ospf)# router-id Loopback0
RP/0/RP0/CPU0:router(config-ospf)# area 1
RP/0/RP0/CPU0:router(config-ospf-ar)# interface POS0/2/0/0
```

By using the Data Object, you can define the OSPF process and instance that you are interested in and ensure that it is started. This is shown in the following example:

```
my $ospf_process = Configuration->OSPF->ProcessTable->Process(InstanceName => '1');
$ospf_process->Start->set_data('true');
```

The following example shows how to set up the router ID for the process:

```
$ospf_process->DefaultVRF->RouterID->InterfaceID->set_data("Loopback0");
```

The following example shows how to set up the area that this interface will be a part of:

```
my $area = $ospf_process->DefaultVRF->AreaTable->
    Area(IPAddressID => '0.0.0.0', IntegerID => 1, AreaIDFormat => 'integer');
$area->Running->set_data('true');
```

The following example shows how to configure the interface:

```
$area->NameScopeTable->NameScope(Interface => "POS0/2/0/0")->Running->
    set_data('true');
```

Getting a List of the Usernames That are Configured on the Router

You may want to get a list of the usernames that are configured on the router, but without all other information that is displayed by the CLI command **show aaa userdb**. The following example shows where you would use the `get_keys` function:

```
my $response = Configuration->AAA->UsernameTable->get_keys;
my @keys = $response->get_keys;
```

R3.3 Beta Draft—Cisco Confidential Information

You could use the resulting array however you want; for example, to display your own compact list of usernames. This is shown in the following example:

```
print "Usernames configured on the system:\n";
foreach my $key (@keys) {
    print "$key->{Name}\n";
}
```

Finding the IP Address of All Interfaces That Have IP Configured

The example shows how to use the `find_data` function of the Data Object interface to find every occurrence of a particular leaf object that matches the combination of key values and wildcards that are specified in the hierarchy.

The code that is needed is almost identical to that which would be used to get the IP address of a particular interface, except that the `Name` key to the interface configuration table is specified as a wildcard, and the function calls the `find_data` method rather than `get_data` method. This is shown in the following example:

```
my $if_conf_table = Configuration->InterfaceConfigurationTable;
my $response = $if_conf_table->
    InterfaceConfiguration(Active => 'act', wildcard => 'Name')->
    IPV4Network->Addresses->Primary->find_data;
```

The XML response can be extracted from the returned response object using the `to_string` method. In addition, the XML response can be examined programmatically by extracting the DOM tree representation from the response object using the `get_dom_tree` method.

Adding an Entry to the Access Control List

The following commands show how to add an entry to an access control list (ACL) to block a particular source IP address:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# ipv4 access-list user1
RP/0/RP0/CPU0:router(config-ipv4-acl)# deny ip host 1.2.3.4 any
```

The following commands show how to add an ACL to the inbound traffic of an interface:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# interface POS0/2/0/0
RP/0/RP0/CPU0:router(config-if)# ipv4 access-group user1 in
```

You can also perform the tasks using the Perl Data Object API. The code acts as though the specified ACL already exists and the last sequence number in the list is already known. The last sequence number for the new entry can be easily calculated. The following example shows how to define the relevant tables that you are interested in:

```
my $acl_table = Configuration->IPV4_ACLAndPrefixList->AccessListTable;
my $if_conf_table = Configuration->InterfaceConfigurationTable;
```

R3.3 Beta Draft—Cisco Confidential Information

The following example shows how to add a new entry to the ACL. (This request sets all other items in an access list entry to null.)

```
my $acl = $acl_table->AccessList(Name => 'user1');
$error = $acl->AccessListEntryTable->
    AccessListEntry(SequenceNumber => '50')->
        set_data(SourceAddress => '1.2.3.4',
            Grant => 'Deny',
            Protocol => 'IP');
```

The following example shows how to add the ACL to the interface:

```
my $interface = $if_conf_table->
    InterfaceConfiguration(Active => 'act', Name => 'POS0/2/0/0');
$error = $interface->IPv4PacketFilter->Inbound->set_data('user1');
```

Denying Access to a Set of Interfaces from a Particular IP Address

The intended use of the script is to quickly and easily block a particular IP address from gaining access to the router on whichever interfaces that you choose, for example, a security threat. This is a good example of a script that retrieves some existing configuration data. Based on the information, some new configuration is applied to the router.

In practice, the set requests are all sent in one request as described in the following list:

- The interfaces in which the new ACL entry is to be applied, the IP address to block, and the name of the new ACL (if needed) are entered by you when prompted. If desired, `all` can be specified instead of listing all interfaces on the system.
- The router is queried to see which access control lists are defined, and what the current entries are to find the last sequence number in each list.
- The router is queried to see which inbound ACLs are assigned to each interface, if any.



Note The request retrieves only the specific configuration that is desired, which can be done by using the CLI. In addition, it makes use of a wildcard for the Name key of the interface table to get the required data for all interfaces, which can also be done by using the CLI.

- If any of those interfaces did not already have an ACL assigned to it, a new ACL is created and assigns it to those interfaces.
- The new ACL entry is added to each of the existing ACLs that were assigned to one of the interfaces in question, and to the new ACL if there is one.



Note The example could be easily extended to block more than one IP address, or to apply the new ACL entry to multiple routers at one time. The script can be found in the `examples/acl/deny_access.pl` file.

Each configuration item is set with an individual call to the `set_data` function of the Data Object interface. Usually, this would result in many separate XML requests. Because the batching API is used, the configuration is set using a single XML request to maximize efficiency.

R3.3 Beta Draft—Cisco Confidential Information

Performing Actions Whenever Certain Events Occur for Notifications

The sample, which demonstrates how to use the Cisco IOS XR Perl Notification and Alarm API, shows how to perform an action whenever a certain event occurs. In particular, it informs someone through e-mail (network administrator) about the events listed in [Table 15-8](#).

Table 15-8 **List of Events**

Event	Description
Interfaces going up/down	When the event occurs, an e-mail is sent informing the recipient of the interface, which router the interface is on, and the new state.
Configuration change	When the running configuration on the router is changed, an e-mail is sent informing the recipient that the event occurred. The configuration change event includes the commitID of the latest commit, the location of a file that contains the commit changes in XML format, and a readable version of the commit changes.

The following steps for the script are described:

1. Registers for alarms for the two relevant types, which are determined by specifying the Group and Code fields, and records the two returned registration IDs.
2. Enters an event loop in which the script calls the alarm_receive function to get the next alarm from the session and calls the relevant handler determined by the registration ID of the alarm.

For change in configuration, differences are retrieved from the router using the same management session that is used for receiving alarms. The XML response is stored in a local file with each commit being stored in a separate file. A readable version of the differences, which is created automatically by using the data object in a string context, is included in the e-mail.

An e-mail, is sent to the specified address, which may be regular e-mail or a message sent to a pager. This is not practical for a long message (for example, a configuration change), but would be well-suited to a single-line message similar to the interface up/down case.

In essence, actions taken when an event occurs are not limited to sending e-mails. A script could do just about anything in response to an event; for example, performing actions or changing configuration on the router. In addition, a script could register to receive notifications from more than one router, which gives it the ability to know the state of a whole network and perform actions accordingly.

The script can be found in the examples/notification/notification.pl file.



Note

The script uses the Perl module Mail::Send, which must be installed to use it.

Operational Examples

Some examples can be used to retrieve operational data from the router. These examples all make use of the Perl Data Object API due to the ease with which requests can be formed, and the flexibility of having access to a Perl representation of the response data and the XML form.

Some examples are very simple, such as retrieving all data in a particular table, and requires only a couple lines of code. Other examples involve getting data from more than one place and combining the data somehow. An example script is provided.

R3.3 Beta Draft—Cisco Confidential Information

There are some scripts that give examples of how to display the retrieved data in different ways. First are some examples of producing a textual output similar to the corresponding CLI command. These use the Data Object interface because of the ease with which the desired data can be extracted from the Perl representation of the response.

Some examples can be used to transform an XML response into an HTML table for easy viewing in a web browser. These examples take full advantage of having the response data in XML format. HTML can be produced very easily from XML using the style sheet transformation language XSLT.

Retrieving the Operational Information for All Interfaces on the Router

The following example shows how to retrieve a single table of data, which can be done very easily by using the Data Object interface `get_entries` function. It can be done all in one line (or rather one statement that has to be split over multiple lines). For clarity and to take advantage of error checking however, it is best to do the retrieval in stages.

To retrieve the operational information for all interfaces on the router, perform the following steps:

-
- Step 1** Define the key for the data node that you are interested in (for example, primary RP), as shown in the following example:
- ```
my $data_node = {RPLocation => {Rack => 0, Slot => RP0, Instance => 'CPU0'}};
```
- Step 2** Define the table and what contents to retrieve, as shown in the following example:
- ```
my $interface_table = Operational->InterfaceProperties->DataNodeTable->
    DataNode($data_node)->SystemView->InterfaceTable;
```
- Step 3** Call the `get_entries` function on the table, as shown in the following example:
- ```
my $response = $interface_table->get_entries;
```
- Step 4** Check to see if an error occurred. If not, retrieve the entries, as shown in the following example:
- ```
if (!defined($result->get_error)) {
    my @entries = $result->get_entries;
    # Now do something with the entries...
}
```
-

Retrieving the Link State Database for a Particular Level for ISIS

Another example is to retrieve a single table of data, which is accomplished by using the `get_entries` function. The data that is retrieved corresponds roughly to that displayed by the CLI command **show isis database level <level no>** and split up into three stages. You will notice that the last two stages are exactly the same.

To retrieve the link state database for a particular level for ISIS, perform the following steps:

-
- Step 1** Define the table from which you want to get data from, as shown in the following example:
- ```
my $table = Operational->ISIS->InstanceTable->Instance(InstanceID => 1)->
 LevelTable->LevelInstance(Level => "Level1")->LSPTTable;
```
- Step 2** Call the `get_entries` function on the table, as shown in the following example:
- ```
my $response = $table->get_entries;
```
-

R3.3 Beta Draft—Cisco Confidential Information

Step 3 Check to see if an error occurred. If not, retrieve the entries, as shown in the following example:

```
if (!defined($result->get_error)) {
    my @entries = $result->get_entries;
    # Now do something with the entries...
}
```

Getting a List of All Interfaces on the System

The simple example shows how to use the `get_keys` function to get a list of the items in a table without getting all the other associated data with them, which cannot be done using the CLI **show** commands.

To get a list of all interfaces on the system, perform the following steps:

Step 1 Define the table, as shown in the following example:

```
my $interface_table = Operational->InterfaceProperties->DataNodeTable->
    DataNode(RPLocation => {Rack => 0, Slot => RP0, Instance => "CPU0"})->
    SystemView->InterfaceTable;
```

Step 2 Call the `get_keys` function and check for errors, as shown in the following example:

```
my $response = $interface_table->get_keys;
if ($result->get_error) {
    die "Error in get_keys: $result->get_error";
}
```

The list of interfaces is returned now as an array from `$response->get_keys` to carry out an action for each interface.

The following example shows how to print it:

```
foreach my $if ($result->get_keys) {
    print $if->{Name} . "\n";
}
```

Retrieving the Combined Interface and IP Information for Each Interface

This is a more complicated example of retrieving operational data as it gets the data from more than one place and combines that data in some way. The `get_ip_interfaces()` function retrieves the operational state for each interface and the IPv4 information for each interface that has IPv4 information.

These two sets of information are combined into one table so that the data is easily accessed by a script that wants to display the data. This is exactly the information that is used by the CLI command **show ip interfaces**.

The function is a good example of how the use of XML makes scripts robust to changes in the underlying data. For example, if new data items are added to the tables, or names of items change, the function still works. The function can be found in the `examples/interfaces/get_ip_interfaces.pm` file within the toolkit installation directory.

R3.3 Beta Draft—Cisco Confidential Information

Listing the Hostname and Interface for Each ISIS Neighbor

You can call the `get_keys` function on the ISIS neighbor table that returns the interface name and system ID for each neighbor. The system ID is an internal value that uniquely identifies the neighbor, but it is not very useful as a displayed value. However, a second table called the *hostname table* provides a mapping from the system ID to the actual hostname (amongst other things). The hostname is displayed by the CLI `show` command **show isis neighbors**, rather than by the system ID. Thus, combining the data from these tables, you can produce a list of the hostname and interface name for each neighbor.

The `list_isis_neighbors` function example resides in the `examples/isis/list_isis_neighbors.pm` file. The function calls the `get_keys` function on the `NeighborTable` object, which produces a list of the system ID and interface for each neighbor. Then, it calls the `get_entries` function on the `HostnameTable` object, and maps the resulting table into a hash, which provides a mapping from system ID to hostname. Finally, the mapping is used to create a new array containing a list of the hostname and interface for each neighbor.

As an example of using the `list_isis_neighbors` function, the following code prints the hostname and interface for each neighbor for ISIS instance 1:

```
require '<toolkit inst dir>/examples/isis/list_isis_neighbors.pm';
my @neighbors = list_isis_neighbors(1);
print "Interface:      Hostname:\n";
foreach my $nbr (@neighbors) {
    printf("%-20s%s\n", $nbr->{Interface}, $nbr->{Hostname});
}
```

Recreating the Output of the show ip interfaces CLI Command

The example shows how to write an easily customized script for displaying information retrieved from a router.

The script gets the required data by calling the `get_ip_interfaces()` function. For details, see [“Retrieving the Combined Interface and IP Information for Each Interface” section on page 15-163](#). The script goes through each entry in the table and picks particular data items, and displays them in a custom format that is the same format as the original **show** CLI command.

The display function easily can be customized by removing sections when data is no longer of interest, adding sections if new data needs to be displayed, or changing the way particular data is displayed. You can create your own version of the **show ip interfaces** CLI command.

The function is clearly more dependent on the names of the data items that are returned and their formats than the underlying `get_ip_interfaces()` function. Because of XML, the function still works if extra items are added to or removed from tables that are not currently being displayed.

The script can be found in the `/examples/interfaces/show_ip_interfaces.pl` file within the toolkit installation directory.

Producing a Textual Output Similar to the show bgp neighbors CLI Command

This is another example of displaying data retrieved from the router in a custom format and again in the same style as the original **show** CLI command. The data-retrieval part of the script is simple and uses the `get_entries` function, as shown in the following example:

```
my $response = Operational->BGP->NeighborTable->get_entries;
```

The script goes through each of the returned entries and calls the `print_neighbor_info` function to display the details of the neighbor.

R3.3 Beta Draft—Cisco Confidential Information

The function shows how easily the required items can be accessed and displayed, and how to ignore information in which you are not interested (for example, AF-specific information, which uses the Data Object interface).

The script can be found in the examples/bgp/show_bgp_neighbors.pl file within the toolkit installation directory.

Displaying Tabular XML Data in a Generic HTML Table Using XSLT

HTML is one of the most useful ways to display data. The HTML format has many useful features, such as the ease in which it can display formatted data in a platform-independent way, and the ability to add links for easy navigation.

For data that takes the form of a list of records (for example, a table), an HTML table is a natural way to display it. A sample function has been provided that uses XSLT to transform a table of data in XML format into an HTML table.

The function is generic. You can pass as an argument the name of the table that you want to display, and the script automatically tries to produce the best HTML table it can. If the table is simple (for example, each field in the table has exactly one value), the output should be a good representation of the data.

If the structure of the data is more complicated (for example, certain fields contain multiple subfields or even subtables within the table), the contents of these subfields or subtable appears within one field and probably will not be very useful.

The function can be found in the examples/common/xml_to_html_table.pm file within the toolkit installation directory. The XSL file it uses to do the transformation is in examples/common/xml_to_html_table.xsl.



Note

The XML::LibXSLT module must be installed to use the example.

To use the function, you may want to display the operational data for each interface on the router as follows:

Step 1 Use the require statement to specify the name of the module, as shown in the following example:

```
require "xml_to_html_table.pm"; # may need to specify a path here
```

Step 2 Retrieve the information in XML format. The following example uses the Data Object interface to do this:

```
my $data_node_table = Operational->InterfaceProperties->DataNodeTable;
my $interface_table = $data_node_table->DataNode(RPLocation =>
    {Rack => 0, Slot => 0, Instance => "CPU0"})->SystemView->InterfaceTable;
my $response_string = $interface_table->find_data->to_string;
```

Step 3 Transform the XML to HTML, as shown in the following example:

```
xml_to_html_table($response_string, $html_file, "InterfaceTable");
```

The example can be found in the examples/interfaces/generic_interface_props_table.pl file.

The resulting HTML file contains a table with a row for each interface and a column for each field of data. Clearly, the function is best suited for tables in which the number of fields is small enough that they all fit the screen. However, the main drawback to using the generic function is that the display

R3.3 Beta Draft—Cisco Confidential Information

format of the fields is identical to XML, which may not be desired—as is the case with the Type, State, and Line State fields in the interface properties example. For more information, see [“Displaying the Interface State in a Customized HTML Table” section on page 15-166](#).

Displaying the Interface State in a Customized HTML Table

In many cases, the generic HTML table example does not display the information quite as you want it. For example, you may want to display only some data items for each table entry and change the display format of certain items.

The example produces an HTML containing the State and Line State for each interface and ignores all other data in the interface table. The following enhancements are provided over the generic HTML table:

- Only the fields of interest are displayed, and they are displayed in the order desired rather than the order that they appear in XML.
- The State and Line State fields are converted from their numeric values to text values that are easier to understand.
- Color-coding is added so that important information, such as an interface being down, stands out.
- The interfaces are sorted, so any interfaces that are down appear before those that are up—which makes it easy to spot problems without having to scroll down a long list.

The transformation is again done using XSLT. The XML::LibXSLT module must be installed to run the example. This means that the Perl script is very short and most of the work is done in the XSL file, which can be easily modified to customize the format of the displayed data, or extended to display more of the information returned in the XML.

The request in the example is stored as preformed XML to demonstrate the use of the basic Perl XML API, but the script could easily have been written using the Data Object API to form the request.

The example can be found in the examples/interfaces/interface_props_table.pl file. The XSL style sheet can be found in the examples/interfaces/interface_props_table.xsl file.

Displaying the BGP Neighbor Operational Data in a Complex HTML Format

This is an example of displaying data that does not conform naturally to a simple table format. The data displayed corresponds roughly to the command-**show bgp neighbors**, for which the output has an entry for each BGP neighbor and within each entry a subtable of information exists for each address family. Because the intended use of the script is for monitoring, the only values shown are operational rather than configurational.

Unlike the previous example, the script uses the Perl Data Object API to create the request that avoids having to write to any XML, which makes it quicker to write and easier to understand and maintain. However, the response format used is still XML and is needed to transform into HTML using XSLT.

The layout of the HTML output has a structure similar to that of the **show** command, with each BGP neighbor entry consisting of a selection of items laid out in a logical way, which includes the address family information that is displayed in a simple subtable.

The benefits of having the information in HTML are as follows:

- The format of bold headings makes the information clearer.
- The layout is much easier to control using tables, because they automatically adjust themselves to fit the information contained in them and the available space on the screen.

R3.3 Beta Draft—Cisco Confidential Information

To facilitate navigation of the neighbor list, a separate HTML page is created that contains a simple summary table, with one entry for each neighbor. Each neighbor in the table has a link, which when clicked jumps to the neighbor's entry in the main table.

When the script is run, a session is created on the router that repeatedly polls the router for the latest information at regular intervals by updating the HTML files each time. Each of the two HTML files causes the web browser to automatically refresh them at the same regular interval, so the values on screen are automatically kept up to date. Ideally, the script would be modified to run as a CGI script on a web server, so that you can just open the web page (from any machine that has access) and not have to start the script first.

The script can be found in the `examples/bgp/bgp_neighbor_table_html.pl` file. The summary page produced is `examples/bgp/bgp_neighbor_table_summary.html`, and the main page is `examples/bgp/bgp_neighbor_table.html`.

**Note**

The XML::LibXSLT module must be installed to run the example.

R3.3 Beta Draft—Cisco Confidential Information

Sample BGP Configuration

The following excerpt displays the relevant portions of the command-line interface (CLI) configuration, which is used as a basis for the Border Gateway Protocol (BGP) examples contained within this document:

```
router bgp 3
  timers bgp 60 180
  bgp router-id 10.1.0.1
  bgp update-delay 55
  bgp cluster-id 10.1.0.2
  bgp graceful-restart purge-time 300
  default-information originate
  bgp graceful-restart restart-time 180
  bgp log neighbor changes disable
  default-metric 10
  bgp graceful-restart stalepath-time 300
  bgp graceful-restart
  bgp as-path-loopcheck
  socket send-buffer-size 131072
  bgp bestpath med always
  bgp bestpath compare-routerid
  bgp bestpath med missing-as-worst
  socket receive-buffer-size 131072
  address-family ipv4 unicast
    distance bgp 140 145 150
    bgp dampening 1 1400 1800 2
    bgp scan-time 30
    network 10.100.1.0/24
    network 10.100.2.0/24
    aggregate-address 10.100.0.0/16 summary-only
    redistribute connected route-map MATCH_ONE_CONNECT
    redistribute static route-map MATCH_ONE_STATIC
  exit
  address-family ipv4 multicast
    distance bgp 120 125 130
    maximum-paths 6
    bgp dampening 2 2400 2800 3
    bgp scan-time 40
    network 10.10.1.0/24
    network 10.10.2.0/24
    aggregate-address 80.100.0.0/16 summary-only
    redistribute connected
  exit
  .
  .
  .
neighbor 10.0.101.1
  remote-as 1
```

R3.3 Beta Draft—Cisco Confidential Information

```

ebgp-multihop 255
address-family ipv4 unicast
  send-community-ebgp
  route-map EBGp_IN_1 in
exit
address-family ipv4 multicast
  send-community-ebgp
  route-map EBGp_IN_1 in
exit
exit
neighbor 10.0.101.2
  remote-as 2
  ebgp-multihop 255
  address-family ipv4 unicast
    send-community-ebgp
    capability orf prefix-list receive
    route-map EBGp_IN_1 in
  exit
  address-family ipv4 multicast
    send-community-ebgp
    route-map EBGp_IN_1 in
  exit
exit
neighbor 10.0.101.3
  remote-as 3
  address-family ipv4 unicast
    route-map IBGP_IN_1 in
  exit
  address-family ipv4 multicast
    route-map IBGP_IN_1 in
  exit
exit
neighbor 10.0.101.4
  remote-as 4
  ebgp-multihop 255
  address-family ipv4 unicast
    route-map EBGp_IN_2 in
  exit
  address-family ipv4 multicast
    route-map EBGp_IN_2 in
  exit
exit
neighbor 10.0.101.5
  remote-as 5
  ebgp-multihop 255
  address-family ipv4 unicast
    route-map EBGp_IN_3 in
  exit
  address-family ipv4 multicast
    route-map EBGp_IN_3 in
  exit
exit
neighbor 10.0.101.6
  remote-as 6
  ebgp-multihop 255
  address-family ipv4 unicast
    prefix-list orf in
    capability orf prefix-list both
  exit
  address-family ipv4 multicast
    prefix-list orf in
  exit
exit
neighbor 10.0.101.7

```

R3.3 Beta Draft—Cisco Confidential Information

```
remote-as 7
ebgp-multihop 255
address-family ipv4 unicast
  prefix-list orf in
  capability orf prefix-list send
exit
address-family ipv4 multicast
  prefix-list orf in
exit
exit
neighbor 10.0.101.8
  remote-as 8
  ebgp-multihop 255
  address-family ipv4 multicast
  exit
exit
.
.
.
exit
```

R3.3 Beta Draft—Cisco Confidential Information

A

AAA authentication, authorization, and accounting. A network security service that provides the primary framework to set up access control on a router or access server. AAA is an architectural framework and modular means of configuring three independent, but closely related security functions in a consistent manner.

B

BGP Border Gateway Protocol. A routing protocol used between autonomous systems. It is the routing protocol that makes the Internet work. BGP is a distance vector routing protocol that carries connectivity information and an additional set of BGP attributes. These attributes allow for a rich set of policies for deciding the best route to reach a given destination.

Border Gateway Protocol See BGP.

C

CLI command-line interface.

CORBA Common Object Request Broker Architecture. Specification that provides the standard interface definition between OMG-compliant objects. CORBA allows applications to communicate with one another, no matter where they are located or who has designed them.

CWI Craft Works Interface. Remote web-based monitoring, configuration, fault management, and troubleshooting system for the router.

E

eBGP external Border Gateway Protocol. BGP sessions are established between routers in different autonomous systems. eBGPs communicate among different network domains.

EGP Exterior Gateway Protocol. Internet protocol for exchanging routing information between different autonomous systems. EGP is an obsolete protocol that was replaced by BGP. See also *BGP*.

extensible markup language See XML.

R3.3 Beta Draft—Cisco Confidential Information

G

Gbps Gigabits per second. The amount of data that can be sent in a fixed amount of time. 1 gigabit = 2^{30} bits, 1,073,741,824 bits.

H

HTTP Hypertext Transfer Protocol. Used by web browsers and web servers to transfer files, such as text and graphic files. The Hypertext Transfer Protocol (HTTP) is the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. Relative to the TCP/IP suite of protocols (which are the basis for information exchange on the Internet), HTTP is an application protocol.

Hypertext Transfer Protocol See HTTP.

I

IDL Interface Definition Language. External client applications use CORBA IDL to exchange XML encoded request and response streams with the CORBA XML agent running on the router

IIOP Internet Inter-ORB Protocol (IIOP). The protocol used for accessing objects across the Internet.

IOR Interoperable Object Reference. A bundle of data that is used by a CORBA application to determine how to connect to a server and send requests to an object.

IOS XR The Cisco operating system used on the router.

L

line card See modular services card. Line cards are now referred to as MSCs in the router.

LR logical router. A routing system can be partitioned into several logical routers, each of which is managed independently. The terms *router* and *LR* are used interchangeably in this document.

M

modular services card Module in which the ingress and egress packet processing and queueing functions are carried out in the router architecture. Up to 16 MSCs are installed in a line card chassis; each MSC must have an associated physical layer interface module (PLIM) (of which there are several types to provide a variety of physical interfaces). The MSC and PLIM mate together on the line card chassis midplane.

MSCs are also referred to as *line cards*.

R3.3 Beta Draft—Cisco Confidential Information

MPLS-TE Multiprotocol Label Switching traffic engineering.

MSC See modular services card.

N

node A card installed and running in a Cisco routing system. In the Cisco XR 12000 Series Router, nodes are identified by slot number (for example, node 1).

R

router Network layer device that uses one or more routing metrics to determine the optimal path along which network traffic should be forwarded. Routers forward packets from one network to another based on network layer information.

running configuration The router configuration in effect. Although, the user can save multiple versions of the router configuration for future reference, only one copy of the running configuration is in the router at any given time. An explicit commit operation must be performed to make changes to or update the running configuration on the router.

S

software configuration A list of packages activated for a particular node. A software configuration consists of a boot package and additional feature packages.

SSH Secure Shell.

SSL Secure Socket Layer.

startup configuration The router configuration designated to be applied on next router startup.

switchover A switch between the active and standby cards; the old active card may be dead prior to switchover (death of the active card is one of the causes for the switchover). Also known as failover.

system reload Reload of a Cisco router node.

system restart Soft reset of a Cisco router node. This involves restarting all the processes running on that node.

T

TAC Cisco Technical Assistance Center

target configuration The current Cisco IOS XR running configuration plus the autonomous changes made to that configuration by a user. The target configuration is promoted to the running configuration by means of the **commit** command.

R3.3 Beta Draft—Cisco Confidential Information

Tbps	Terabits per second = 1,000,000,000,000 (1 trillion) bits per second. The amount of data that can be sent in a fixed amount of time.
Telnet	Standard terminal emulation protocol in the TCP/IP protocol stack. Telnet is used for remote terminal connection, enabling users to log in to remote systems and use resources as if they were connected to a local system. Telnet is defined in RFC 854.
Terabyte	A unit of computer memory or data storage capacity equal to 1,024 gigabytes (2^{40} bytes). Approximately 1 trillion bytes.

X

XML	extensible markup language. A standard maintained by the World Wide Web Consortium (W3C) that defines a syntax that lets you create markup languages to specify information structures. Information structures define the type of information (for example, subscriber name or address), not how the information looks (bold, italic, and so on). External processes can manipulate these information structures and publish them in a variety of formats. XML allows you to define your own customized markup language.
XML agent	A process on the router that is sent XML requests by XML clients and is responsible for carrying out the actions contained in the request and returning an XML response back to the client. The XML agent for CORBA is an example of an XML agent provided on the router.
XML client	An external application that sends an XML request to the router and receives XML responses to those requests.
XML operation	A portion of an XML request that specifies an operation that the XML client would like the XML agent to perform.
XML operation provider	The Cisco router code that carries out a particular XML operation including parsing the operation XML, performing the operation, and assembling the operation XML response.
XML request	An XML document sent to the router, containing a number of requested operations to be carried out.
XML response	The response to an XML request.
XML schema	An XML document specifying the structure and possible contents of XML elements that can be contained in an XML document.

R3.3 Beta Draft—Cisco Confidential Information

Symbols

- <Action> 60
- <AdminAction> 60
- <AdminOperational> 60
- <Alarm> 107
- <Clear> 23
- <CLI> 24
- <CLI> tag 53, 55, 89
- <ClientName> 50, 51
- <Comment> 50
- <Commit> 23, 36
 - Comment attribute 36
 - errors 38
 - IgnoreOtherSessions attribute 37
 - KeepFailedConfig attribute 36
 - Label attribute 36
 - Mode attribute 36
 - Replace attribute 37
 - Rollback 45
- <Commit> operation 39
- <CommitId> tag 37, 45, 50
- <Configuration> 60
- <Configuration/ > tag 72
- <Delete> 23, 38, 52, 59
 - AAA privileges 97
 - native data operations 65
- <Delete/ > tag 69
- <destination> 133
- <EBGPMultiHopMaxHopCount> 76
- <Error> element 98
- <FailedConfig> tag 39
- <File> 33
- <Filter> 85
- <Get> 23, 29, 31, 55, 59
 - AAA privileges 97
 - native data operations 65
 - triggering 64
- <GetConfigurationHistory> 23, 50
 - maximum attribute 50
- <GetConfigurationSession> 23
- <GetConfigurationSessions> 51
- <GetNext> 24
 - IteratorID 91
- <GetVersionInformation> 23
- <HoldTime> 64
- <Label> 50
- <Line> 50, 51
- <Load> 23, 33, 39, 58
- <Lock> 23, 29
- <LockHeld> 52
- <LoopbackCheck> 76
- <Naming> tag 64
- <Operational> 55, 60
- <Previous> 46
- <Register> 107
- <RemoteAS> 76
- <Response>
 - IteratorID 91
- <Rollback> 23, 37, 45, 46
- <Save> 23, 33, 35
- <SessionId> 51
- <Set> 23, 38, 53, 59
 - AAA privileges 97
 - native data operations 65
- <Since> 52
- <Timestamp> 50
- <Unlock> 23, 40
- <Unlock/ > 40
- <UserId> 50, 51
- <version> 133

A

- AAA (authentication, authorization, and accounting)
 - authorization 97
 - definition 18
 - login 27

R3.3 Beta Draft—Cisco Confidential Information

- options
 - ASCII authentication [119](#)
 - default [119](#)
 - security (perl scripting toolkit) [132](#)
- ACL (Access Control List)
 - CLI commands [159](#)
 - entry, add [159](#)
 - inbound traffic [159](#)
 - list [160](#)
 - perl data object API [159](#)
- add_neighbors_to_group.pl file [156](#)
- alarm_deregister function [150](#)
- alarm_operations.xsd [128](#)
- alarm_receive function [150](#)
- alarm_register function [150](#)
- alarms
 - deregistration [108](#)
 - event notification [57](#)
 - filter criteria, types of [107](#)
 - notification [109](#)
 - registration [107](#)
 - tags, types of [109](#)
- API (application programming interface)
 - perl data object [132](#)
 - perl notification/alarm [132](#)
 - perl XML
 - concept [132](#)
 - configuration examples [154](#)
 - operational examples [161](#)
- arguments, management session
 - connection_timeout [135](#)
 - host [135](#)
 - interactive [134](#)
 - password [135](#)
 - port [135](#)
 - prompt [135](#)
 - response_timeout [135](#)
 - ssh_version [135](#)
 - transport [135](#)

- use_command_line [134](#)
- username [135](#)
- ASCII authentication option [119](#)
- Atomic mode [36](#)

B

- BASE package common schemas [128](#)
- batch_send method [148](#)
- batch_start method [148](#)
- batch API
 - batch_send method [148](#)
 - batch_start method [148](#)
 - usage [147](#)
- batched requests [24](#)
- BestEffort [36](#)
- BGP (Border Gateway Protocol)
 - CLI commands [155](#)
 - configuration [169](#)
 - data object interface [155](#)
 - get request [55](#)
 - neighbor
 - add list [155](#)
 - members, display [156](#)
 - set description [155](#)
- bgp_neighbor_table_html.pl file [167](#)
- Border Gateway Protocol
 - See* BGP [55](#)
- browse, target configuration [29](#)

C

- cerrno [115](#)
- ChangedConfig [30](#)
- chmod command [154](#)
- CircuitType object [157](#)
- Cisco-IOS_XR-Perl-Scripting-Toolkit-.tar.gz file [133](#)
- Clear tag [23](#)

R3.3 Beta Draft—Cisco Confidential Information

- CLI (command-line interface)
 - CLI-based scripts [17](#)
 - defined [18](#)
 - operations [24](#)
 - cli_operations.xsd [128](#)
 - CLI command
 - encapsulated [19, 24, 89](#)
 - show [60](#)
 - show aaa userdb [158](#)
 - show bgp neighbors [166](#)
 - show ip interfaces [163, 164](#)
 - show isis database level [162](#)
 - show isis neighbors [164](#)
 - show rollback points [50](#)
 - xml agent tty [123](#)
 - XML and [90](#)
 - ClientID attribute [22](#)
 - client session
 - commit operation [38](#)
 - limitation [27](#)
 - CLI tag [24](#)
 - commit
 - changes [41](#)
 - database [37](#)
 - identifier [45](#)
 - Commit tag [23](#)
 - commit target configuration example [138](#)
 - common_datatypes.xsd [128](#)
 - common datatype definitions [128](#)
 - Common Object Request Broker Architecture
 - See* CORBA [17](#)
 - component-specific schemas [127, 128](#)
 - Comprehensive Perl Archive Network
 - See* CPAN [133](#)
 - config_clear function example [139](#)
 - config_cli() function example [140](#)
 - config_commit () function [138](#)
 - config_get_history() function example [139](#)
 - config_get_sessions function example [139](#)
 - config_load_failed function example [139](#)
 - config_rollback() function example [139](#)
 - config_save() function example [139](#)
 - config_services_operations.xsd [128](#)
 - Configuration change event [161](#)
 - configuration change notification [58](#)
 - Configuration function [157](#)
 - configuration history [28](#)
 - Configuration Manager [19, 23, 98](#)
 - and error reporting [115](#)
 - Configuration services [19, 23, 98](#)
 - configuration session information [28](#)
 - connection_timeout argument [135](#)
 - container [71, 73](#)
 - Content attribute [71, 82](#)
 - CORBA (Common Object Request Broker Architecture)
 - event notification channel [57](#)
 - support, type of [17](#)
 - Count argument [146](#)
 - Count attribute [71, 84](#)
 - Craft Works Interface [18](#)
 - CurrentConfig [30](#)
 - custom filters [71](#)
 - CWI (Craft Works Interface)
 - See* Craft Works Interface [18](#)
-
- ## D
- data, display how to
 - example [148](#)
 - get_data function [148](#)
 - data objects
 - create [142](#)
 - operation methods [144](#)
 - schema version [143](#)
 - data operation methods, management session [149](#)
 - debug facility
 - definition, types of [152](#)
 - disable [152](#)

R3.3 Beta Draft—Cisco Confidential Information

enable [152](#)
 insert message [152](#)
 overview [151](#)
 debug option [135](#)
 declaration
 attributes [21](#)
 tag [20, 21](#)
 default option [119](#)
 delete_data method
 definition [147](#)
 example [147](#)
 Delete tag [23](#)
 deny_access.pl file [160](#)
 dependencies [63](#)
 deregistering, alarms [108](#)
 display_neighbor_group_members.pl file [156](#)
 documentation, perl data object
 definition items [141](#)
 overview [141](#)
 Document Type Definition [127](#)
 DOM (Data Object Model)
 example [137](#)
 tree type [137](#)
 DTD (Document Type Definition)
 See document type definition [127](#)

E

element, null value [64](#)
 encoding (UTF-8), XML [21](#)
 error attributes [112, 113](#)
 ErrorCode [112](#)
 ErrorMessage [112](#)
 error object, methods
 get_code [138](#)
 get_dom_node [138](#)
 get_element [138](#)
 get_message [138](#)
 to_string [138](#)

error reporting
 nonexistent data [68](#)
 types of [111](#)
 event notification [57](#)
 extensible markup language
 See XML [ix](#)

F

files, perl scripting toolkit
 add_neighbors_to_group.pl [156](#)
 bgp_neighbor_table.html.pl [167](#)
 Cisco-IOS_XR-Perl-Scripting-Toolkit-.tar.gz [133](#)
 deny_access.pl [160](#)
 display_neighbor_group_members.pl [156](#)
 generic_interface_props_table.pl [165](#)
 get_ip_interfaces.pm [163](#)
 interface_props_table.pl [166](#)
 interface_props_table.xml [166](#)
 ios_xr_log.txt [153](#)
 list_isis_neighbors.pm [164](#)
 notification.pl [161](#)
 show_bgp_neighbors.pl [165](#)
 show_ip_interfaces.pl [164](#)
 xml_to_html_table.pm [165](#)
 xml_to_html_table.xml [165](#)

filter, criteria types [107](#)

Filter argument [146](#)

find_data function [159](#)

find_data method
 definition [145](#)
 example [145](#)

G

generic_interface_props_table.pl file [165](#)
 get_code method [138](#)
 get_commit_id() method example [138](#)

R3.3 Beta Draft—Cisco Confidential Information

[get_data method](#)
 definition [145](#)
 example [145](#)
[get_dom_node method](#) [138](#)
[get_dom_tree method](#) [159](#)
[get_element method](#) [138](#)
[get_entries function](#) [162, 164](#)
[get_entries method](#)
 definition [146](#)
 example [146](#)
[get_error method example](#) [137](#)
[get_errors method example](#) [137](#)
[get_ip_interfaces\(\) function](#) [163, 164](#)
[get_ip_interfaces.pm file](#) [163](#)
[get_keys function](#) [164](#)
[get_keys method](#)
 definition [145](#)
 example [145](#)
[get_message method](#) [138](#)
[GetConfigurationHistory tag](#) [23](#)
[GetConfigurationSessions tag](#) [23](#)
[GetNext tag operation](#) [23, 24](#)
[Get tag](#) [23](#)
[GetVersionInfo tag](#) [23](#)

H

[hash structure](#)
 definition [140](#)
 example [140](#)
[host argument](#) [135](#)
[HostnameTable object](#) [164](#)
[HTML table](#)
 customize, interface state display [166](#)
 enhancement list [166](#)

[installation, perl scripting toolkit](#)
 directory parameters [133](#)
 procedure [133](#)
[interactive argument](#) [134](#)
[interface_props_table.pl file](#) [166](#)
[interface_props_table.xsl file](#) [166](#)
[interfaces, get list](#)
 examples [163](#)
 procedure [163](#)
[Interfaces going up/down event](#) [161](#)
[InterfaceTable object](#) [157](#)
[ios_xr_log.txt file](#) [153](#)
[IP address, find interfaces](#) [159](#)
[IPv4 address family example](#) [157](#)
[ISIS \(Intermediate System-to-Intermediate System\)](#)
 circuit type, find [157](#)
 CLI commands [156](#)
 hostname and interface, list [164](#)
 instance ID [156](#)
 set up [156](#)
[IteratorID](#) [91](#)

K

[keys, display how to](#)
 example [148](#)
 get_keys function [148](#)

L

[leaf object](#) [74](#)
[link state database, retrieval](#)
 examples [162](#)
 procedure [162](#)
[list_isis_neighbors.pm file](#) [164](#)
[load configuration file example](#) [139](#)
[Load tag](#) [23](#)

R3.3 Beta Draft—Cisco Confidential Information

lock [27, 28](#)
 lock and unlock configuration example [138](#)
 Lock tag [23](#)
 log_file option [135](#)
 logging facility
 arguments, types of [153](#)
 disable [153](#)
 enable [152](#)
 overview [152](#)
 logging option [135](#)

M

make command [133](#)
 make install command [133](#)
 management session
 close
 close()method [136](#)
 script [136](#)
 data operation methods [149](#)
 start
 arguments [134](#)
 create, object type [134](#)
 Match attribute [78](#)
 MergedConfig [30](#)
 modules, perl scripting toolkit [132](#)
 mpls-te task name [99](#)

N

namespace [60](#)
 native_data_common.xsd [128](#)
 native_data_operations.xsd [128](#)
 native data
 access techniques [71](#)
 model, types of [19](#)
 operations [59](#)
 request, nonexistent data [68](#)

tags [23](#)
 native management data model [23](#)
 NET (Network Entity Title) example [156](#)
 nonexistent data [68](#)
 notification.pl file [161](#)
 notifications
 alarms [109](#)
 list of events [161](#)
 steps for script [161](#)
 null value [64](#)

O

object class, hierarchy
 combine [71, 75](#)
 compressed [77](#)
 content [59](#)
 duplicated [76](#)
 nonexistent data [68](#)
 operational [74](#)
 operation information, retrieval
 examples [162](#)
 procedure [162](#)
 operation processing errors [111, 115](#)
 OperationType attribute [31](#)
 operation type tag
 CLI [24](#)
 configuration services [23](#)
 definition [23](#)
 native data [23](#)
 structure, top-level [20](#)
 options, command-line
 debug [135](#)
 log_file [135](#)
 logging [135](#)
 telnet_dump_log [135](#)
 telnet_input_log [135](#)
 OSPF (Open Shortest Path First)
 CLI commands [158](#)

R3.3 Beta Draft—Cisco Confidential Information

configuration [158](#)
 router ID [158](#)
 ouni task name [99](#)

P

password argument [135](#)
 perl scripting toolkit, concepts
 perl data object API [132](#)
 perl notification/alarm API [132](#)
 perl XML API [132](#)
 port argument [135](#)
 privileges, security [97](#)
 prompt argument [135](#)

R

read privileges [97](#)
 registering, alarms [107](#)
 repeat naming information [71, 80](#)
 request
 <Get>
 ChangedConfig [31](#)
 batching [24](#)
 definition [18](#)
 maximum size [22](#)
 minor and major version numbers [22](#)
 repeated naming information [80](#)
 tag [20](#)
 top level structure of [20](#)
 Request Type tag [60](#)
 response
 block size [91](#)
 definition [18](#)
 error reporting [111](#)
 large data retrieval (using iterators) [91](#)
 major and minor version numbers [22](#)
 minimum [22](#)

namespace declaration in [64](#)
 nonexistent data [68](#)
 tag [37](#)
 response_timeout argument [135](#)
 rollback [28](#)
 RollbackOnly attribute [50](#)
 Rollback tag [23](#)
 router administration, operational data [60](#)
 running configuration
 browse [29](#)
 browsing [28](#)
 locking [28](#)
 replacing [28, 52](#)
 target configuration commit [36](#)
 unlocking [28, 40](#)

S

Save tag [23](#)
 schema file organization [128](#)
 schemas, XML [127](#)
 set_data method
 definition [146](#)
 example [146](#)
 Set tag [23](#)
 show_bgp_neighbors.pl file [165](#)
 show_ip_interfaces.pl file [164](#)
 show aaa userdb CLI command [158](#)
 show bgp neighbors CLI command [166](#)
 show ip interfaces CLI command [163, 164](#)
 show isis database level CLI command [162](#)
 show isis neighbors CLI command [164](#)
 Source attribute [29](#)
 SSH
 definition [18](#)
 option [124](#)
 ssh_version argument [135](#)
 system logging message (syslog) [57](#)

R3.3 Beta Draft—Cisco Confidential Information

T

tag

- configuration services operation, types of [23](#)
- XML [20](#)
- XML <Response> [22](#)
- XML API [17](#)
- XML mapping, types of [125](#)

target configuration

- browsing [28](#)
- commit [27, 28, 53](#)
 - syslog [57](#)
- commit record [37](#)
- loading [28](#)
- modified, uncommitted [31](#)
- saving to file [28, 35](#)

TaskGrouping attribute [99](#)

task names

- mpls-te [99](#)
- ouni [99](#)

telnet_dump_log

- argument [153](#)
- option [135](#)

telnet_input_log

- argument [153](#)
- option [135](#)

Telnet option [124](#)

to_string method

- description [138](#)
- example [137](#)
- XML response [159](#)

transport argument [135](#)

transport debug type [152](#)

transport errors [111, 112](#)

triggering a <Get> operation [64](#)

TTY transport

- enable agent, how to [123](#)
- enable session, how to [124](#)
- error code [124](#)

exit, how to [124](#)

options

SSH [124](#)

Telnet [124](#)

U

Unlock tag [23](#)

upgrades, schema file [129](#)

use_command_line argument [134](#)

user debug type [152](#)

username argument [135](#)

usernames, get list [158](#)

V

version, XML [21](#)

W

wildcards [71](#)

World Wide Web Consortium (W3C) XML Schema Language [127](#)

write_file method example [137](#)

write privileges [97](#)

X

XLST

- procedure [165](#)
- tabular XML data, display [165](#)

XML (extensible markup language)

- agent [18, 19](#)
- client [18](#)
- instance [64](#)
- operation [18](#)
- operation provider [18](#)
- parse errors [111, 112](#)
- schema [18](#)

R3.3 Beta Draft—Cisco Confidential Information

- definitions for the native data operation type tags [23](#)
- errors [111, 113](#)
- session [22](#)
- xml_api_common.xsd [128](#)
- xml_api_protocol.xsd [128](#)
- xml_response_parts debug type [152](#)
- xml_to_html_table.pm file [165](#)
- xml_to_html_table.xsl file [165](#)
- xml agent tty CLI command [123](#)
- xml debug type [152](#)
- XML mapping tags [125](#)
- XML request
 - receiving [124](#)
 - sending [124](#)
- XML schemas [127](#)