



Cisco IOS XR XML API Guide

Cisco IOS XR Software Release 4.1

April 2011

Americas 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: 408 527-0883

Text Part Number: OL-24657-01

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.

Cisco and the Cisco Logo are trademarks of Cisco Systems, Inc. and/or its affiliates in the U.S. and other countries. A listing of Cisco's trademarks can be found at www.cisco.com/go/trademarks. Third party trademarks mentioned 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. (1005R)

Any Internet Protocol (IP) addresses used in this document are not intended to be actual addresses. Any examples, command display output, and figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses in illustrative content is unintentional and coincidental.

Cisco IOS XR XML API Guide

© 2011 Cisco Systems, Inc. All rights reserved.



CONTENTS

Preface ix

Changes to This Document ix

Obtaining Documentation and Submitting a Service Request ix

CHAPTER 1

Cisco XML API Overview 1-1

Introduction 1-1

Definition of Terms 1-1

Cisco Management XML Interface 1-2

Cisco XML API and Router System Features 1-3

Cisco XML API Tags 1-3

Basic XML Request Content 1-4

Top-Level Structure 1-4

XML Declaration Tag 1-5

Request and Response Tags 1-5

ResultSummary Tag 1-5

Maximum Request Size 1-6

Minimum Response Content 1-6

Operation Type Tags 1-8

Native Data Operation Tags 1-8

Configuration Services Operation Tags 1-9

CLI Operation Tag 1-9

GetNext Operation Tag 1-9

Alarm Operation Tags 1-10

XML Request Batching 1-10

CHAPTER 2

Cisco XML Router Configuration and Management 2-13

Target Configuration Overview 2-13

Configuration Operations 2-14

Additional Configuration Options Using XML 2-14

Locking the Running Configuration 2-15

Browsing the Target or Running Configuration 2-15

Getting Configuration Data 2-16

Browsing the Changed Configuration 2-17

Loading the Target Configuration 2-19

- Setting the Target Configuration Explicitly 2-20
- Saving the Target Configuration 2-21
- Committing the Target Configuration 2-22
 - Commit Operation 2-22
 - Commit Errors 2-25
 - Loading a Failed Configuration 2-27
- Unlocking the Running Configuration 2-28
- Additional Router Configuration and Management Options Using XML 2-28
 - Getting Commit Changes 2-29
 - Loading Commit Changes 2-30
 - Clearing a Target Session 2-32
 - Rolling Back Configuration Changes to a Specified Commit Identifier 2-33
 - Rolling Back the Trial Configuration Changes Before the Trial Time Expires 2-33
 - Rolling Back Configuration Changes to a Specified Number of Commits 2-34
 - Getting Rollback Changes 2-35
 - Loading Rollback Changes 2-36
 - Getting Configuration History 2-38
 - Getting Configuration Commit List 2-41
 - Getting Configuration Session Information 2-43
 - Clear Configuration Session 2-44
 - Replacing the Current Running Configuration 2-45
 - Clear Configuration Inconsistency Alarm 2-46

CHAPTER 3

Cisco XML Operational Requests and Fault Management 3-49

- Operational Get Requests 3-49
- Action Requests 3-50
 - Cisco XML and Fault Management 3-51
 - Configuration Change Notification 3-51

CHAPTER 4

Cisco XML and Native Data Operations 4-53

- Native Data Operation Content 4-53
 - Request Type Tag and Namespaces 4-54
 - Object Hierarchy 4-54
 - Main Hierarchy Structure 4-55
 - Dependencies Between Configuration Items 4-58
 - Null Value Representations 4-58
 - Operation Triggering 4-58
 - Native Data Operation Examples 4-59
 - Set Configuration Data Request: Example 4-60

Get Request: Example	4-62
Get Request of Nonexistent Data: Example	4-63
Delete Request: Example	4-65
GetDataSpaceInfo Request Example	4-66

CHAPTER 5

Cisco XML and Native Data Access Techniques	5-67
Available Set of Native Data Access Techniques	5-67
XML Request for All Configuration Data	5-68
XML Request for All Configuration Data per Component	5-68
XML Request for All Data Within a Container	5-69
XML Request for Specific Data Items	5-71
XML Request with Combined Object Class Hierarchies	5-72
XML Request Using Wildcarding (Match Attribute)	5-75
XML Request for Specific Object Instances (Repeated Naming Information)	5-79
XML Request Using Operation Scope (Content Attribute)	5-82
Limiting the Number of Table Entries Returned (Count Attribute)	5-83
Custom Filtering (Filter Element)	5-85
XML Request Using the Mode Attribute	5-86

CHAPTER 6

Cisco XML and Encapsulated CLI Operations	6-91
XML CLI Command Tags	6-91
CLI Command Limitations	6-92

CHAPTER 7

Cisco XML and Large Data Retrieval	7-93
Iterators	7-93
Usage Guidelines	7-93
Examples Using Iterators to Retrieve Data	7-94
Large Response Division	7-97
Terminating an Iterator	7-97
Throttling	7-98
CPU Throttle Mechanism	7-99
Memory Throttle Mechanism	7-99
Streaming	7-99
Usage Guidelines	7-99

CHAPTER 8

Cisco XML Security	8-101
Authentication	8-101
Authorization	8-101

- Retrieving Task Permissions 8-102
- Task Privileges 8-102
- Task Names 8-103
- Authorization Failure 8-104
- Management Plane Protection 8-104
 - Inband Traffic 8-104
 - Out-of-Band Traffic 8-104
- VRF 8-105
- Access Control List 8-105

CHAPTER 9

Cisco XML Schema Versioning 9-107

- Major and Minor Version Numbers 9-107
- Run-Time Use of Version Information 9-108
 - Placement of Version Information 9-109
 - Version Lag with the AllowVersionMismatch Attribute Set as TRUE 9-110
 - Version Lag with the AllowVersionMismatch Attribute Set as FALSE 9-111
 - Version Creep with the AllowVersionMismatch Attribute Set as TRUE 9-112
 - Version Creep with the AllowVersionMismatch Attribute Set as FALSE 9-113
- Retrieving Version Information 9-113
- Retrieving Schema Detail 9-115

CHAPTER 10

Alarms 10-117

- Alarm Registration 10-117
- Alarm Deregistration 10-118
- Alarm Notification 10-119

CHAPTER 11

Error Reporting in Cisco XML Responses 11-121

- Types of Reported Errors 11-121
 - Error Attributes 11-122
 - Transport Errors 11-122
 - XML Parse Errors 11-122
 - XML Schema Errors 11-123
 - Operation Processing Errors 11-125
 - Error Codes and Messages 11-126

CHAPTER 12**Summary of Cisco XML API Configuration Tags 12-127****CHAPTER 13****XML Transport and Event Notifications 13-129**

TTY-Based Transports 13-129

Enabling the TTY XML Agent 13-129

Enabling a Session from a Client 13-129

Sending XML Requests and Receiving Responses 13-130

Configuring Idle Session Timeout 13-130

Ending a Session 13-130

Errors That Result in No XML Response Being Produced 13-130

Dedicated Connection Based Transports 13-131

Enabling the Dedicated XML Agent 13-131

Enabling a Session from a Client 13-131

Sending XML Requests and Receiving Responses 13-132

Configuring Idle Session Timeout 13-132

Ending a Session 13-132

Errors That Result in No XML Response Being Produced 13-132

SSL Dedicated Connection based Transports 13-132

Enabling the SSL Dedicated XML Agent 13-133

Enabling a Session from a Client 13-133

Sending XML Requests and Receiving Responses 13-133

Configuring Idle Session Timeout 13-133

Ending a Session 13-134

Errors That Result in No XML Response Being Produced 13-134

CHAPTER 14**Cisco XML Schemas 14-135**

XML Schema Retrieval 14-135

Common XML Schemas 14-136

Component XML Schemas 14-136

Schema File Organization 14-136

Schema File Upgrades 14-137

CHAPTER 15**Network Configuration Protocol 15-139**

Starting a NETCONF Session 15-139

Ending a NETCONF Agent Session 15-140

Starting an SSH NETCONF Session 15-140

Ending an SSH NETCONF Agent Session 15-141

Configuring a NETCONF agent 15-141

- Limitations of NETCONF in Cisco IOS XR 15-142
 - Configuration Datastores 15-142
 - Configuration Capabilities 15-142
 - Transport (RFC4741 and RFC4742) 15-142
 - Subtree Filtering (RFC4741) 15-142
 - Protocol Operations (RFC4741) 15-144
 - Event Notifications (RFC5277) 15-145

CHAPTER 16

Cisco IOS XR Perl Scripting Toolkit 16-147

- Cisco IOS XR Perl Scripting Toolkit Concepts 16-148
- Security Implications for the Cisco IOS XR Perl Scripting Toolkit 16-148
- Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit 16-148
- Installing the Cisco IOS XR Perl Scripting Toolkit 16-149
- Using the Cisco IOS XR Perl XML API in a Perl Script 16-150
- Handling Types of Errors for the Cisco IOS XR Perl XML API 16-150
- Starting a Management Session on a Router 16-150
- Closing a Management Session on a Router 16-152
- Sending an XML Request to the Router 16-152
- Using Response Objects 16-153
- Using the Error Objects 16-154
- Using the Configuration Services Methods 16-154
- Using the Cisco IOS XR Perl Data Object Interface 16-157
 - Understanding the Perl Data Object Documentation 16-158
 - Generating the Perl Data Object Documentation 16-158
 - Creating Data Objects 16-159
 - Specifying the Schema Version to Use When Creating a Data Object 16-161
 - Using the Data Operation Methods on a Data Object 16-161
 - get_data Method 16-161
 - find_data Method 16-162
 - get_keys Method 16-162
 - get_entries Method 16-163
 - set_data Method 16-163
 - delete_data Method 16-164
 - Using the Batching API 16-164
 - batch_start Method 16-164
 - batch_send Method 16-165
 - Displaying Data and Keys Returned by the Data Operation Methods 16-165
 - Specifying the Session to Use for the Data Operation Methods 16-166

Cisco IOS XR Perl Notification and Alarm API	16-166
Registering for Alarms	16-166
Deregistering an Existing Alarm Registration	16-167
Deregistering All Registration on a Particular Session	16-167
Receiving an Alarm on a Management Session	16-167
Using the Debug and Logging Facilities	16-168
Debug Facility Overview	16-168
Logging Facility Overview	16-169
Examples of Using the Cisco IOS XR Perl XML API	16-170
Configuration Examples	16-171
Setting the IP Address of an Interface	16-171
Configuring a Simple BGP Neighbor	16-172
Adding a List of Neighbors to a BGP Neighbor Group	16-172
Displaying the Members of Each BGP Neighbor Group	16-173
Setting Up ISIS on an Interface	16-173
Finding the Circuit Type That is Currently Configured for an Interface for ISIS	16-173
Configuring a New Instance, Area, and Interface for OSPF	16-175
Getting a List of the Usernames That are Configured on the Router	16-175
Finding the IP Address of All Interfaces That Have IP Configured	16-175
Adding an Entry to the Access Control List	16-176
Denying Access to a Set of Interfaces from a Particular IP Address	16-176
Configuring a New Static Route Entry	16-177
Operational Examples	16-177
Retrieving the Operational Information for All Interfaces on the Router	16-178
Retrieving the Link State Database for a Particular Level for ISIS	16-178
Getting a List of All Interfaces on the System	16-179
Retrieving the Combined Interface and IP Information for Each Interface	16-179
Listing the Hostname and Interface for Each ISIS Neighbor	16-180
Recreating the Output of the show ip interfaces CLI Command	16-180
Producing a Textual Output Similar to the show bgp neighbors CLI Command	16-180
Displaying Tabular XML Data in a Generic HTML Table Using XSLT	16-181
Displaying the Interface State in a Customized HTML Table	16-182
Displaying the BGP Neighbor Operational Data in a Complex HTML Format	16-182
Performing Actions Whenever Certain Events Occur	16-183
Sample BGP Configuration	17-185

GLOSSARY

INDEX



Preface

The XML application programming interface (API) is available for use on any Cisco platform running Cisco IOS XR software. This document describes the XML API provided to developers of external management applications. The XML interface provides a mechanism for 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 preface contains these sections:

- [Changes to This Document, page ix](#)
- [Obtaining Documentation and Submitting a Service Request, page ix](#)

Changes to This Document

[Table 1](#) lists the technical changes made to this document since it was first published.

Table 1 **Changes to This Document**

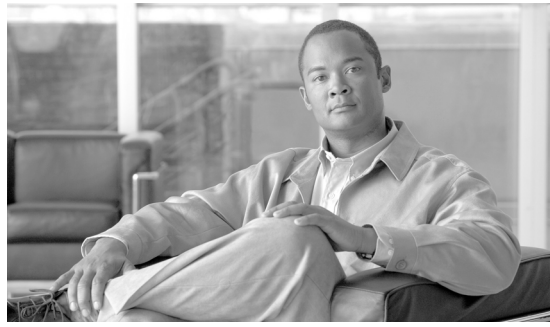
Revision	Date	Change Summary
OL-24657-01	April 2011	Initial release of this document.

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, submitting a service request, and gathering additional information, see the monthly *What's New in Cisco Product Documentation*, which also lists all new and revised Cisco technical documentation, at:

<http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html>

Subscribe to the *What's New in Cisco Product Documentation* as a Really Simple Syndication (RSS) feed and set content to be delivered directly to your desktop using a reader application. The RSS feeds are a free service and Cisco currently supports RSS Version 2.0.



CHAPTER 1

Cisco XML API Overview

This chapter contains these sections:

- [Introduction, page 1-1](#)
- [Cisco Management XML Interface, page 1-2](#)
- [Cisco XML API and Router System Features, page 1-3](#)
- [Cisco XML API Tags, page 1-3](#)

Introduction

This *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 management application developers write client applications to interact with the Cisco XML infrastructure on the router, and to use the Management XML API to build custom end-user interfaces for configuration and information retrieval and display.

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

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.

Customers use a variety of vendor-specific 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 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.

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.
SSH	Secure Shell.
SSL	Secure Sockets Layer.
XML	Extensible markup language.
XML agent	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	External application that sends XML requests to the router and receives XML responses to those requests.
XML operation	Portion of an XML request that specifies an operation that the XML client wants the XML agent to perform.
XML operation provider	Code that carries out a particular XML operation including parsing the operation XML, performing the operation, and assembling the operation XML response.
XML request	XML document sent to the router containing a number of requested operations to be carried out.
XML response	Response to an XML request.
XML schema	XML document specifying the structure and possible contents of XML elements that can be contained in an XML document.

Cisco Management XML Interface

These topics, which are covered in detail in the sections that follow, outline information about 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
- Configuring the router using:
 - the two-stage “target configuration” mechanism provided by the configuration manager
 - features such as locking, loading, browsing, modifying, saving, and committing the configuration
- Accessing the operational data of the router with XML

- Working with native management data object class hierarchies to:
 - represent native data objects in XML
 - use techniques, including the use of wildcards and filters, for structuring XML requests that access the management data of interest,
- Encapsulating CLI commands in XML
- Error reporting to the client application
- Using iterators for large scale data retrieval
- Handling event notifications with XML
- Enforcing authorization of client requests
- Versioning of XML schemas
- Generation and packaging of XML schemas
- Transporting options that enable corresponding XML agents on the router
- Using the Cisco IOS XR Perl Scripting Toolkit to manage a Cisco IOS XR router

Cisco XML API and Router System Features

Using the XML API, an external client application sends XML encoded management requests to an XML agent running on the router. The XML API readily supports available transport layers including terminal-based protocols such as Telnet, Secure Shell (SSH), dedicated-TCP connection, and Secure Sockets Layer (SSL) dedicated TCP connection.

Before an XML session is established, the XML transport and XML agent must be enabled on the router. For more information, see [Chapter 13, “XML Transport and Event Notifications.”](#)

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

- Native data access (get, set, delete, and so on) using the native management data model.
- Configuration services for advanced configuration management through the Configuration Manager.
- Traditional CLI access where 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 sends the XML encoded response stream on to the client.

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 these sections:

- [Basic XML Request Content, page 1-4](#)
- [XML Declaration Tag, page 1-5](#)
- [Operation Type Tags, page 1-8](#)

- [XML Request Batching, page 1-10](#)

Basic XML Request Content

This section describes the specific content and format of 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 begins with an XML declaration tag followed by a response tag, one or more operation type tags, and a result summary tag with an error count. Each request contains operation tags for each supported operation type; these operation type tags can be repeated. The operation type tags contained in the response corresponds to those contained in the client request.

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-specific response data returned here
    .
    .
    .
  </Operation>
  <ResultSummary ErrorCount="0"/>
</Response>
```



Note

All examples in this document are formatted with line breaks and white space to aid readability. Actual XML request and response streams that are exchanged with the router do not include such line breaks and white space characters. This is 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 is specified in the corresponding request. Note The encoding attribute is optional.

Request and Response Tags

Following the XML declaration tag, the client application must enclose each request stream within a pair of <Request> start and </Request> end tags. Also, the system encloses each XML response within a pair of <Response> start and </Response> end tags. Major and minor version numbers 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 request and response streams. A client application issues a request after which, the router returns a response. The client then issues 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 ignored by the router.

ResultSummary Tag

The system adds a <ResultSummary> tag immediately before the </Response> end tag to indicate the overall result of the operation performed. This tag contains the attribute ErrorCount to indicate the total number of errors encountered. A value of 0 indicates no errors. If applicable, the ItemNotFound or ItemNotFoundBelow attributes are also included. See Table 1-3 for explanations of these attributes.

Sample XML Response with ResultsSummary Tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
.
.
<ResultSummary ErrorCount="0" ItemNotFoundBelow="true"/>
</Response>
```

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 13, “XML Transport and Event Notifications.”](#)

Minimum Response Content

If a <Set> or <Delete> request has nothing to return, the router returns the original request and an appropriate empty operation type tag. The minimum response returned by the router with a single operation <Set> or <Delete> and no result data, is shown in these examples:

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/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

If a <Get> request has nothing to return, the router returns the original request with an ItemNotFound attribute at the <Get> level.

If a <Get> request has some ‘not found’ elements to return, the router returns the original request with an ItemNotFoundBelow attribute at the <Get> level. For each requested element that is not found, the router returns a NotFound attribute at the element level. For each requested element that is present, it returns the corresponding data.

[Table 1-3](#) defines the attributes when the <Get> request does not have any elements to return.

Table 1-3 Attributes for Elements Not Found

Attribute	Description
ItemNotFound	Empty response at the <Get> level.
ItemNotFoundBelow	Response with some requested elements that are not found at the <Get> level.
NotFound	Requested element is not found at the element level.

Sample XML Request from Client Application (ItemNotFound)

```
<?xml version="1.0"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
```

```

<Configuration>
  <InterfaceConfigurationTable>
    <InterfaceConfiguration>
      <Naming>
        <Active>act</Active>
        <InterfaceName>Loopback1</InterfaceName>
      </Naming>
    </InterfaceConfiguration>
  </InterfaceConfigurationTable>
</Configuration>
</Get>
</Request>

```

Sample XML Minimum Response from a Router (ItemNotFound)

```

<?xml version="1.0"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get ItemNotFound="true">
    <Configuration>
      <InterfaceConfigurationTable MajorVersion="4" MinorVersion="2">
        <InterfaceConfiguration NotFound="true">
          <Naming>
            <Active>act</Active>
            <InterfaceName>Loopback1</InterfaceName>
          </Naming>
        </InterfaceConfiguration>
      </InterfaceConfigurationTable>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0" ItemNotFound="true"/>
</Response>

```

Sample XML Request from Client Application (ItemNotFoundBelow)

```

<?xml version="1.0"?>
<Request MajorVersion="1" MinorVersion="0">
<Get>
  <Configuration>
    <InterfaceConfigurationTable>
      <InterfaceConfiguration>
        <Naming>
          <Active>act</Active>
          <InterfaceName>Loopback0</InterfaceName>
        </Naming>
        <Description/>
        <Shutdown/>
        <IPV4Network/>
      </InterfaceConfiguration>
    </InterfaceConfigurationTable>
  </Configuration>
</Get>
</Request>

```

Sample XML Minimum Response from a Router (ItemNotFoundBelow)

```

<?xml version="1.0"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get ItemNotFoundBelow="true">
    <Configuration>
      <InterfaceConfigurationTable MajorVersion="4" MinorVersion="2">
        <InterfaceConfiguration>

```

```

    <Naming>
      <Active>act</Active>
      <InterfaceName>Loopback0</InterfaceName>
    </Naming>
    <Description>desc-loop0</Description>
    <Shutdown NotFound="true"/>
    <IPV4Network MajorVersion="5" MinorVersion="1">
      <Addresses>
        <Primary>
          <IPAddress>1.1.1.1</IPAddress>
          <Netmask>255.255.0.0</Netmask>
        </Primary>
      </Addresses>
    </IPV4Network>
  </InterfaceConfiguration>
</InterfaceConfigurationTable>
</Configuration>
</Get>
<ResultSummary ErrorCount="0" ItemNotFoundBelow="true"/>
</Response>

```

Operation Type Tags

Following the <Request> tag, the client application must specify the operations 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 1-4](#) describes the native data operation tags.

Table 1-4 Native Data Operation Tags

Native Data Tag	Description
<Get>	Gets the value of one or more configuration, operational, or action data items.
<Set>	Creates or modifies one or more configuration or action data items.
<Delete>	Deletes one or more configuration data items.
<GetVersionInfo>	Gets the major and minor version numbers of one or more components.
<GetDataSpaceInfo>	Retrieves native data branch names.

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 1-5](#) describes the configuration services operation tags.

Table 1-5 Configuration Services Operation Tags

Tag	Description
<Lock>	Locks the running configuration.
<Unlock>	Unlocks the running configuration.
<Load>	Loads the target configuration from a binary file previously saved using the <Save> tag.
<Save>	Saves the target configuration to a binary file.
<Commit>	Promotes the target configuration to the running configuration.
<Clear>	Aborts or clears the current target configuration session.
<Rollback>	Rolls back the running configuration to a previous configuration state.
<GetConfigurationHistory>	Gets a list of configuration events.
<GetConfigurationSessions>	Gets a list of the user sessions currently configuring the box.
<Get ConfigurationCommitList>	Gets a list of commits that were made to the running configuration and can be rolled back.
<ClearConfigurationSession>	Clears a particular configuration session.
<ClearConfigurationInconsistency>	Clears a configuration inconsistency alarm.

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 <GetNext> operation, see [Chapter 7, “Cisco XML and Large Data Retrieval.”](#)

Alarm Operation Tags

The <Alarm> operation tag registers, unregisters, and receives alarm notifications. [Table 1-6](#) lists the alarm operation tags.

Table 1-6 List of Alarm Operation Tags

Tag	Description
<Register>	Registers to receive alarm notifications.
<Unregister>	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”](#)).

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, in a case where operations 1 through 3 are in the request, even if operation 2 fails, operation 3 is attempted.

If you want to perform two or more <Get> operations, and if the first one might 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 [Chapter 5, “Cisco XML and Native Data Access Techniques.”](#)

This 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>
</Request>
```



```
      .  
      .  
      .  
      .  
      .  
      </Operational>  
    </Get>  
  <Unlock/>  
  <ResultSummary ErrorCount="0" />  
</Response>
```




CHAPTER 2

Cisco XML Router Configuration and Management

This chapter reviews the basic 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 that is being modified 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 these sections:

- [Target Configuration Overview, page 2-13](#)
- [Configuration Operations, page 2-14](#)
- [Additional Router Configuration and Management Options Using XML, page 2-27](#)

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 dedicated TCP, Telnet, Secure Shell (SSH) connection, or SSL dedicated 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.

**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 these configuration options from the client application to configure or modify the router with XML:

- [Locking the Running Configuration](#), page 2-14
- [Browsing the Target or Running Configuration](#), page 2-15
 - [Getting Configuration Data](#), page 2-15
- [Browsing the Changed Configuration](#), page 2-16
- [Loading the Target Configuration](#), page 2-19
- [Setting the Target Configuration Explicitly](#), page 2-20
- [Saving the Target Configuration](#), page 2-21
- [Committing the Target Configuration](#), page 2-22
 - [Loading a Failed Configuration](#), page 2-26
- [Unlocking the Running Configuration](#), page 2-27

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 ErrorMessage attributes that indicates the cause of the lock failure.

This 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/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

These conditions apply when the running configuration is locked:

- 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 only read 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 tags. 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.

Getting Configuration Data

Table 2-1 describes the Source options.

Table 2-1 Source Options

Option	Description
ChangedConfig	Reads 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	Reads from the current active running configuration. This option corresponds to the CLI command show configuration running .
MergedConfig	Reads 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 could 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 .

Table 2-1 Source Options (continued)

Option	Description
CommitChanges	Reads from the commit database for the specified commit ID. This operation corresponds to the CLI command show configuration commit changes .
RollbackChanges	Reads from a set of rollback changes. This operation corresponds to the CLI command show configuration rollback-changes .

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

This example shows a <Get> request used 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="18" 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>
  <ResultSummary ErrorCount="0"/>
</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.

This 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 these 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>
          <AS>
            <Naming>
              <AS>0</AS>
            </Naming>
            <FourByteAS>
              <Naming>
                <AS>3</AS>
              </Naming>
              <DefaultVRF>
                <Global>
                  <DefaultMetric>10</DefaultMetric>
                </Global>
              </DefaultVRF>
            </FourByteAS>
          </AS>
        </BGP>
      </Configuration>
    </Set>
    <Delete>
      <Configuration>
        <BGP>
          <AS>
            <Naming>
              <AS>0</AS>
            </Naming>
            <FourByteAS>
              <Naming>
                <AS>3</AS>
              </Naming>
              <DefaultVRF>
                <BGPEntity>
                  <NeighborTable>
                    <Neighbor>
                      <Naming>
                        <NeighborAddress>
                          <IPV4Address>10.0.101.8</IPV4Address>
                        </NeighborAddress>
                      </Naming>
                    </Neighbor>
                  </NeighborTable>
                </BGPEntity>
              </DefaultVRF>
            </FourByteAS>
          </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>
  <ResultSummary ErrorCount="0"/>
</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/>
    </Configuration>
  </Get>
</Request>

```

Sample Secondary XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="ChangedConfig" OperationType="Set">
      <BGP MajorVersion="30" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
          <BGPRunning>true</BGPRunning>
          <DefaultVRF>
            <Global>
              <DefaultMetric>
                10
              </DefaultMetric>
            </Global>
          </DefaultVRF>
        </FourByteAS>
      </AS>
    </BGP>
  </Configuration>
  <Configuration Source="ChangedConfig" OperationType="Delete">
    <BGP MajorVersion="30" MinorVersion="0">
      <AS>
        <Naming>
          <AS>0</AS>
        </Naming>
        <FourByteAS>
          <Naming>
            <AS>3</AS>
          </Naming>

```

```

    <DefaultVRF>
      <BGPEntity>
        <NeighborTable>
          <Neighbor>
            <Naming>
              <NeighborAddress>
                <IPv4Address>
                  10.0.101.8
                </IPv4Address>
              </NeighborAddress>
            </Naming>
          </Neighbor>
        </NeighborTable>
      </BGPEntity>
    </DefaultVRF>
  </FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0" />
</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 configuration file saved using CLI is not loadable with XML <Load>. The configuration should have been saved using the XML <Save> operation. Using the <Load> operation is strictly optional. It can be used alone or with the <Set> and <Delete> operations, as described in the section [“Setting the Target Configuration Explicitly”](#) section on page 2-20.

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 contains the ErrorCode and ErrorMessage attributes that indicate the cause of the load failure.

This 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>
<ResultSummary ErrorCount="0"/>
</Response>

```

See also the [“Setting the Target Configuration Explicitly”](#) section on page 20.

Setting the Target Configuration Explicitly

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



Note

There are no 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 can result in the 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 ErrorMessage 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.

This 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 these 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)# 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>
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
          <DefaultVRF>
            <Global>
              <DefaultMetric>10</DefaultMetric>
              <GlobalTimers>
                <Keepalive>60</Keepalive>
                <HoldTime>180</HoldTime>
              </GlobalTimers>
            </Global>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
</Request>

```



```

        </GlobalTimers>
    </Global>
    </DefaultVRF>
    </FourByteAS>
    </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>
    <ResultSummary ErrorCount="0"/>
</Response>

```

To replace a portion of the configuration, the client application should use a <Delete> operation to remove the unwanted portion of the 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 the [“Replacing the Current Running Configuration” section on page 2-44](#).

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 also contains the ErrorCode and ErrorMessage attributes that indicate the cause of the failure.

This 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>
<ResultSummary ErrorCount="0" />
</Response>

```

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

Attribute	Description
Mode	Use the Mode attribute to specify whether the target configuration should be committed on an Atomic or a BestEffort basis. In the case of a commit with the Atomic option, the entire configuration in the target area is committed only if the application of all of 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 too, the errors are 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 is 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.

Table 2-2 Commit Operation Attributes (continued)

Attribute	Description
Confirmed	Use the Confirmed attribute as a commit request, which sends the target configuration to a trial commit. The confirmed request has a value of 30 to 300 seconds. If the user sends a commit request without the Confirmed attribute within the specified period, the changes are committed; otherwise, the changes are rolled back after the specified period is over. If the user sends a commit request again with the Confirmed attribute, the target configuration is sent to the trial commit.
Replace	Use this boolean attribute to specify whether the commit operation should replace the entire configuration running on the router with the contents of the target configuration buffer. The default value for Replace is false. The Replace attribute should be used with caution. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">  <p>Caution The new configuration must contain the necessary configuration to maintain the XML session, for example, “xml agent” or “xml agent tty” along with the configuration for the management interface. Otherwise, the XML session is terminated.</p> </div>
IgnoreOtherSessions	Use this boolean attribute to specify whether the commit operation should be allowed to go through without an error when 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.

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.

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

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"/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

This example shows a request to commit for a 50-second period. The request corresponds to the **commit confirmed 50** CLI command.

Sample XML Client Request to Commit for a 50-second Period

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Commit Confirmed="50"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Commit Confirmed="50"
    CommitID="1000000075"/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

These 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, such as “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 (using the `<CommitID>` tag).
- 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 the [“Replacing the Current Running Configuration” section on page 2-44](#).
- 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.
- To confirm the commit with the Confirmed attribute, the user has to send an explicit `<Commit/>` without the Confirmed attribute or send a `<Commit/>` without the “Confirmed” attribute along with any other configurations.

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), all of the failed configuration is returned in the `<Commit>` response along with the appropriate `ErrorCode` and `ErrorMsg` 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.

This 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 these 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>
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>4</AS>
            </Naming>
            <DefaultVRF>
              <Global>
                <DefaultMetric>10</DefaultMetric>
              </Global>
            </DefaultVRF>
          </FourByteAS>
        </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>
  <ResultSummary ErrorCount="0"/>
</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="30" MinorVersion="0">
        <AS>
          <Naming>
            <AS>4</AS>
          </Naming>
          <FourByteAS>
            <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>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Commit>
<ResultSummary ErrorCount="1"/>
</Response>

```

For more information, see the [“Loading a Failed Configuration”](#) section on page 2-26.

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

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

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>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <FailedConfig/>
  </Load>
  <Get>
    <Configuration Source="ChangedConfig" OperationType="Set">
      <BGP MajorVersion="30" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>4</AS>
            </Naming>
          <BGPRunning>
            true
          </BGPRunning>
          <DefaultVRF>

```

```

        <Global>
          <DefaultMetric>
            10
          </DefaultMetric>
        </Global>
      </DefaultVRF>
    </FourByteAS>
  </AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</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 <Unlock/> tag. If the unlock operation fails, the response can also contain the ErrorCode and ErrorMessage attributes that indicate the cause of the unlock failure.

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

Sample XML Client Request to Unlock the Running Configuration

```

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

```

Sample XML Response from the Router

```

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

```

Additional Router Configuration and Management Options Using XML

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

- [Getting Commit Changes, page 2-28](#)
- [Loading Commit Changes, page 2-29](#)
- [Clearing a Target Session, page 2-31](#)
- [Rolling Back Configuration Changes to a Specified Commit Identifier, page 2-32](#)
- [Rolling Back the Trial Configuration Changes Before the Trial Time Expires, page 2-32](#)
- [Rolling Back Configuration Changes to a Specified Number of Commits, page 2-33](#)

- [Getting Rollback Changes, page 2-34](#)
- [Loading Rollback Changes, page 2-35](#)
- [Getting Configuration History, page 2-37](#)
- [Getting Configuration Commit List, page 2-40](#)
- [Getting Configuration Session Information, page 2-42](#)
- [Clear Configuration Session, page 2-43](#)
- [Replacing the Current Running Configuration, page 2-44](#)
- [Clear Configuration Inconsistency Alarm, page 2-45](#)

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 and subsequently forwarded to any registered configuration agents.

Table 2-3 describes the event notification.

Table 2-3 Event Notification

Notification	Description
userid	Name of the user who performed the commit operation.
timestamp	Date and time of the commit.
commit	Unique ID associated with the commit.

This example shows a configuration change notification:

```
RP/0/1/CPU0:Jul 25 18:23:21.810 : config[65725]: %MGBL-CONFIG-6-DB_COMMIT :
Configuration committed by user 'lab'. Use 'show configuration commit changes
1000000001' to view the changes
```

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

The client application can read a set of commit changes using the <Get> operation along with the <Configuration> request type tag when it includes the Source attribute option CommitChanges. One of the additional attributes, either ForCommitID or SinceCommitID, must also be used to specify the commit identifier or commit label for which the commit changes should be retrieved.

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

Sample XML Request to Show Specified Commit Changes Using the ForCommitID Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" ForCommitID="1000000075"/>
  </Get>
</Request>
```


Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" ForCommitID="100000075"
      OperationType="....">
      .
      .
      .
      changed config returned here
      .
      .
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>
```

This example shows the use of the `SinceCommitID` attribute to show the commit changes made since a specific commit. This request corresponds to the **show configuration commit changes since 100000072** CLI command.

Sample XML Request to Show Specified Commit Changes Using the `SinceCommitID` Attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" SinceCommitID="100000072"/>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" SinceCommitID="100000072">
      OperationType="....">
      .
      .
      .
      changed config returned here
      .
      .
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</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 configuration.

If the load succeeds, the response contains both the `Load` and `CommitChanges` tags. If the load fails, the response also contains the `ErrorCode` and `ErrorMsg` attributes indicating the cause of the load failure.

This 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 **load commit changes 100000072** CLI command.

Sample XML Request to Load Commit Changes with the ForCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <ForCommitID>100000072</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>100000072</ForCommitID>
    </CommitChanges>
  </Load>
  <ResultSummary ErrorCount="0" />
</Response>
```

This 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 **load commit changes since 100000072** CLI command.

Sample XML Request to Load Commit Changes with the SinceCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <SinceCommitID>100000072</SinceCommitID>
    </CommitChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Load>
    <CommitChanges>
      <SinceCommitID>100000072</SinceCommitID>
    </CommitChanges>
  </Load>
  <ResultSummary ErrorCount="0" />
</Response>
```

This 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 **load commit changes last 4** CLI command.

Sample XML Request to Load Commit Changes with the 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/>
    <Previous>4</Previous>
  </CommitChanges>
</Load>
  <ResultSummary ErrorCount="0"/>
</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.

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

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/>
  <ResultSummary ErrorCount="0"/>
</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 ErrorMessage 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)

Attribute	Description
Label	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	User-specified comment to be associated with the rollback in the router commit database.

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

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>
  <ResultSummary ErrorCount="0"/>
</Response>
```



Note

The commit identifier can also be obtained by using the <GetConfigurationHistory> operation described in the section [“Getting Configuration History”](#) section on page 2-37.

Rolling Back the Trial Configuration Changes Before the Trial Time Expires

When the user sends a commit request with the Confirmed attribute, a trial configuration session is created. If the user then sends a confirmed commit, the trial configuration changes are committed. If the user wants to roll back the trial configuration changes before the trial time expires, the user can use the <Rollback> operation.

**Note**

No optional attributes can be used when `<Trial Configuration>` is specified.

This example shows a request to roll back the trial configuration changes:

Sample XML Request to Roll Back the Trial Configuration Before the Trial Time Expires

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

Sample XML Response from the Router

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

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)*

Attribute	Description
Label	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	User-specified comment to be associated with the rollback in the router commit database.

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

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>
  <ResultSummary ErrorCount="0"/>
</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.

This 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 **show configuration rollback-changes to 100000072** CLI command.

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="100000072"/>
  </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="100000072">
      OperationType="....>
      .
      .
      rollback changes returned here
      .
      .
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>
```

This 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 **show configuration rollback-changes last 4** CLI command.

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>
  < ResultSummary ErrorCount="0" />
</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, ToCommidID, or Previous. After the completion of the Load operation, the client application can then modify and commit the rollback changes like with any other configuration.

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

This 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 **load rollback changes 100000072** CLI command.

Sample XML Client to Load Rollback Changes with the ForCommitID tag

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Load>
    <RollbackChanges>
      <ForCommitID>100000072</ForCommitID>
    </RollbackChanges>
  </Load>
</Request>
```

Sample XML Response from the Router

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

This 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 **load rollback changes to 1000000072** CLI command.

Sample XML Client to Load Rollback Changes with the 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>
      <ToCommitID>1000000072</ToCommitID>
    </RollbackChanges>
  </Load>
  <ResultSummary ErrorCount="0" />
</Response>
```

This 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 **load rollback changes last 4** CLI command.

Sample XML Client to Load Rollback Changes with the 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>
      <Previous>4</Previous>
    </RollbackChanges>
  </Load>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Getting Configuration History

The client application uses the <GetConfigurationHistory> operation to get information regarding these configuration events:

- Commit
- Online insertion and removal (OIR) events, also known as remove and replace
- Router shutdown synchronization
- **cfs check** rebuild of persistent configuration from running configuration
- Startup application of admin and SDR configuration, noting alternate configuration fallback specification
- Configuration inconsistency including failed configuration or other similar reasons

Table 2-6 describes the optional attributes available with the <GetConfigurationHistory> operation.

Table 2-6 *Optional Attributes to Get Configuration History*

Attribute	Description
Maximum	Maximum number of entries to be returned from the commit history file. The range of entries that can be returned are from 0 to 1500. If the Maximum attribute is not included in the request, or if the value of the Maximum attribute is greater than the actual number of entries in the commit history file, all entries in the commit history files are returned. The commit entries are returned with the most recent commit history information appearing first in the list.
EventType	Type of event records to be displayed from the configuration history file. If this attribute is not included in the request, all types of event records are returned. The EventType attribute expects one of these values: All, Alarm, CFS-Check, Commit, OIR, Shutdown, or Startup.
Reverse	Reverse attribute has a value of true. If it is specified, the most recent records are displayed first; otherwise, the oldest records are displayed first.
Details	Used to display detailed information. The Detail attribute has a value of either true or false and the default is false.

The <GetConfigurationHistory> operation corresponds to the **show configuration history** CLI command.

This example shows a request to list the information associated with the previous three commits. This request corresponds to the **show configuration commit history first 6 detail** CLI command.

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 EventType="All" Detail="true" Maximum="6"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetConfigurationHistory EventType="All" Detail="true" Maximum="6">
    <EventEntry>
      <Naming>
        <EventName>
          CFS-Check
        </EventName>
      </Naming>
      <Timestamp>
        1300262221
      </Timestamp>
      <Detail>
        <UserID>
          lab
        </UserID>
        <Line>
          vty2
        </Line>
      </Detail>
    </Event Entry>
    <Event Entry>
      <Naming>
        <EventName>
          Commit
        </EventName>
      </Naming>
      <Timestamp>
        1300262224
      </Timestamp>
      <Detail>
        <CommitID>
          1000000627
        </CommitID>
        <UserID>
          lab
        </UserID>
        <Line>
          vty2
        </Line>
        <ClientName>
          CLI
        </ClientName>
      </Detail>
    </Event Entry>
    <Event Entry>
      <Naming>
        <EventName>
          Commit
        </EventName>
      </Naming>
      <Timestamp>
        1300262231
      </Timestamp>
      <Detail>
```

```

        <CommitID>
            1000000628
        </CommitID>
        <UserID>
            lab
        </UserID>
        <Line>
            vty0
        </Line>
        <Client>CLI
        </Client>
    </Detail>
</EventEntry>
<EventEntry>
    <Naming>
        <EventName>
            Commit
        </EventName>
    </Naming>
    <Timestamp>
        1300262239
    </Timestamp>
    <Detail>
        <CommitID>
            1000000629
        </CommitID>
        <UserID>
            lab
        </UserID>
        <Line>
            vty0
        </Line>
        <ClientName>
            CLI
        </ClientName>
    </Detail>
</EventEntry>
<EventEntry>
    <Naming>
        <EventName>
            Commit
        </EventName>
    </Naming>
    <Timestamp>
        1300262246
    </Timestamp>
    <Detail>
        <CommitID>
            1000000630
        </CommitID>
        <UserID>
            lab
        </UserID>
        <Line>
            vty0
        </Line>
        <ClientName>
            CLI
        </ClientName>
    </Detail>
</EventEntry>
<EventEntry>
    <Naming>
        <EventName>

```

```

        Commit
      </EventName>
    <Naming>
    <Timestamp>
      1300262255
    </Timestamp>
    <Detail>
      <CommitID>
        1000000631
      </CommitID>
      <UserID>
        lab
      </UserID>
      <Line>
        vty0
      </Line>
      <ClientName>
        CLI
      </ClientName>
    </Detail>
  </EventEntry>
</GetConfigurationHistory>
<ResultSummary ErrorCount="0" />
</Response>

```

Getting Configuration Commit List

The client application can use the `<GetConfigurationCommitList>` operation to get information regarding the most recent commits to the running configuration.

Table 2-7 describes the information that is returned for each configuration commit session.

Table 2-7 Returned Session Information

Name	Description
<code><CommitID></code>	Unique ID associated with the commit.
<code><Label></code>	(Optional) Label associated with the commit.
<code><User></code>	Name of the user who created the configuration session within which the commit was performed.
<code><Line></code>	Line used to connect to the router for the configuration session.
<code><Client></code>	Name of the client application that performed the commit.
<code><Timestamp></code>	Period of time, in seconds, of the commit.
<code><Comment></code>	(Optional) Comment associated with the commit.
<code><Maximum></code>	(Optional) Maximum number of entries to return from the commit database. If the Maximum attribute is not included in the request or if the Maximum attribute 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 entries appearing first in the list.
<code><Detail></code>	(Optional) Used to get the detailed information about the commit entry. The Detail attribute has the value of true or false and the default value is false.

The <GetConfigurationCommitList> operation corresponds to the **show configuration commit list** CLI command.

This example shows a request to list the information associated with the previous two commits. This request corresponds to the **show configuration commit list 2** CLI command.

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">
  <GetConfigurationCommitList Maximum="2"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetConfigurationHistory Maximum="2">
    <CommitEntry>
      <Naming>
        <CommitID>
          1000000462
        </CommitID>
      </Naming>
      <UserID>
        lab
      </UserID>
      <Line>
        /dev/vty0:node0_0_CPU0
      </Line>
      <ClientName>Rollback
      </Client>
      <Timestamp>
        1303319582
      </Timestamp>
    </CommitEntry>
    <CommitEntry>
      <Naming>
        <CommitID>
          1000000461
        </CommitID>
      </Naming>
      <User>
        lab
      </User>
      <Line>
        /dev/vty0:node0_0_CPU0
      </Line>
      <ClientName>
        XML TTY Agent
      </Client>
      <Timestamp>
        1303318704
      </Timestamp>
    </CommitEntry>
  </GetConfigurationCommitList>
  <ResultSummary ErrorCount="0"/>
</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>	Unique autogenerated ID for the configuration session.
<UserID>	Name of the user who created the configuration session.
<Line>	Line used to connect to the router.
<ClientName>	User-friendly name of the client application that created the configuration session.
<Since>	Date and time of the creation of the configuration session.
<LockHeld>	Boolean operation indicating whether the session has an exclusive lock on the running configuration.

The Detail attribute can be specified with <GetConfigurationSessions>. This attribute specifies whether the detailed information is required. False is the default value.

Table 2-9 describes the additional information that is returned when the Detail attribute is used.

Table 2-9 Returned Session Information with the Detail Attribute

Returned Session Information	Session Information Description
<Process>	Process name
<ProcessID>	Process ID
<Node>	Node ID
<Elapsed>	Session time elapsed, in seconds.

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

Sample XML Request to Get List of Users Configuring the Router

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

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetConfigurationSessions Detail="true">
    <Session>
      <Naming>
        <SessionID>
          00000000-0005f109-00000000
```

```

        </SessionID>
    </Naming>
    <UserID>
        lab
    </UserID>
    <Line>
        con0_0_CPU0
    </Line>
    <Since>
        1303317929
    </Since>
    <LockHeld>
        false
    </LockHeld>
    <TrialSession>
        false
    </TrialSession>
    <Detail>
        <ClientName>
            CLI
        </ClientName>
        <ProcessID>
            389385
        </ProcessID>
        <Process>
            config
        </Process>
        <Node>
            <Rack>
                0
            </Rack>
            <Slot>
                0
            </Slot>
            <Instance>
                CPU0
            </Instance>
        </Node>
        <ElapsedTime>
            2183
        </ElapsedTime>
    </Detail>
    </Session>
</GetConfigurationSessions>
<ResultSummary ErrorCount="0" />
</Response>

```

Clear Configuration Session

The client application can use the `<ClearConfigurationSession>` operation to clear a particular configuration session. The `SessionID` attribute specifies the session to be cleared.

This example shows a request to clear a configuration session. This request corresponds to the **clear configuration sessions 00000000-000a00c9-00000000** CLI command.

Sample XML Request to Get List of Users Configuring the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <ClearConfigurationSession SessionID="00000000-000a00c9-00000000"/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <ClearConfigurationSession SessionID="00000000-000a00c9-00000000"/>

  <ResultSummary ErrorCount="0"/>
</Response>
```

Replacing the Current Running Configuration

A client application replaces the current running configuration on the router with a users configuration file. Performg these operations in sequence:

1. Lock the configuration.
2. 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.
3. Commit the target configuration specifying the Replace attribute with a value of true.

These examples illustrate 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/>
  <ResultSummary ErrorCount="0"/>
</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>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Sample XML Request to Commit the Target Configuration

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

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Commit CommitID="1000000075"/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Clear Configuration Inconsistency Alarm

The client application uses the `<ClearConfigurationInconsistency>` operation to clear a bi-state configuration inconsistency alarm.

If the clear operation is successful, the response contains only the `<ClearConfigurationInconsistency/>` tag. If the clear operation fails, the response also contains the `ErrorCode` and `ErrorMsg` attributes, indicating the cause of the clear failure.

This example shows a request to clear the configuration inconsistency alarm in user mode. This request corresponds to the **clear configuration inconsistency** CLI command.

Sample XML Request to Clear the Configuration Inconsistency Alarm

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

Sample XML Response from the Router

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



CHAPTER 3

Cisco XML Operational Requests and Fault Management

A client application can send an XML request to get router operational information using either a native data <Get> request along with the <Operational> tag, or the equivalent 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 these sections:

- [Operational Get Requests, page 3-49](#)
- [Action Requests, page 3-50](#)

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.”](#)

This 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>
        <Active>
          <DefaultVRF>
            <ProcessInfoTable/>
          </DefaultVRF>
        </Active>
      </BGP>
    </Operational>
  </Get>
</Request>
```

Sample XML Response from the Router

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

```

<Operational>
  <BGP MajorVersion="22" MinorVersion="2">
    <Active>
      <ProcessInfoTable>
        <ProcessInfo>
          <Naming>
            <ProcessID>0</ProcessID>
          </Naming>
          <ProcessInstance>
            0
          </ProcessInstance>
          ....
        more response content here
        ...
      </ProcessInfo>
    </ProcessInfoTable>
  </Active>
</BGP>
</Operational>
</Get>
<ResultSummary ErrorCount="0"/>
</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”.

This example shows an action request to clear the BGP performance statistics information. This request is equivalent to the **clear 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>
        <DefaultVRF>
          <ClearPerformanceStats>true</ClearPerformanceStats>
        </DefaultVRF>
      </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>
  <ResultSummary ErrorCount="0"/>
</Response>
```

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

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>
        <DefaultVRF>
          <ClearDrops>
            <All>true</All>
          </ClearDrops>
        </DefaultVRF>
      </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>
  <ResultSummary ErrorCount="0"/>
</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 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	Name of the user who performed the commit operation.

Table 3-1 *Event Notifications for Configuration Changes (continued)*

Event Notification	Description
timestamp	Date and time of the commit.
commit	Unique ID associated with the commit.

This 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
```

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



CHAPTER 4

Cisco XML and Native Data Operations

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 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 these sections:

- [Request Type Tag and Namespaces, page 4-54](#)
- [Object Hierarchy, page 4-54](#)
- [Dependencies Between Configuration Items, page 4-58](#)
- [Null Value Representations, page 4-58](#)
- [Operation Triggering, page 4-58](#)
- [Native Data Operation Examples, page 4-59](#)

This 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>
```

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>
  <ResultSummary ErrorCount="0" />
</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 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 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.

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, 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.

This 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.

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

<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.

This example shows hierarchy tables:

```
<Configuration>
  <BGP>
    .
    .
    .
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <NeighborAddress>
              <IPV4Address>10.0.101.6</IPV4Address>
            </NeighborAddress>
          </Naming>
          <RemoteAS>
            <AS_XX>
              0
            </AS_XX>
            <AS_YY>
              6
            </AS_YY>
          </RemoteAS>
        </Neighbor>
        <Neighbor>
          <Naming>
            <NeighborAddress>
              <IPV4Address>10.0.101.7</IPV4Address>
            </NeighborAddress>
          </Naming>
          <RemoteAS>
            <AS_XX>
              0
            </AS_XX>
            <AS_YY>
              6
            </AS_YY>
          </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.

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

This example shows table filtering:

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Operational>
      <BGP>
        <Active>
          <VRFTable>
            <VRF>
              <Naming>
                <VRFName>one</VRFName>
              </Naming>
              <NeighborTable>
                <Filter>
                  <BGP_AFFilter>
                    <AFName>IPv4Unicast</AFName>
                  </BGP_AFFilter>
                </Filter>
              </NeighborTable>
            </VRF>
          </VRFTable>
        </Active>
      </BGP>
    </Operational>
  </Get>
</Request>
```

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 4-55. 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.

This 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 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.

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.

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

```
<Neighbor>
  <Timers>
    <KeepAliveInterval>60</KeepAliveInterval>
    <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 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).

This example shows a request to get the confederation peer information for a particular BGP autonomous system. In this example, the <Get> operation is triggered when the <ConfederationPeerASTable/> 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>
      <AS>
        <Naming>
          <AS>0</AS>
        </Naming>
      <FourByteAS>
        <Naming>
          <AS>3</AS>
        </Naming>
      <DefaultVRF>
        <Global>
          <ConfederationPeerASTable/>
        </Global>
      </DefaultVRF>
    </FourByteAS>
  </AS>
</BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="30" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
        <FourByteAS>
          <Naming>
            <AS>3</AS>
          </Naming>
        <DefaultVRF>
          <Global>
            <ConfederationPeerASTable>
              <ConfederationPeerAS>
                <Naming>
                  <AS_XX>0</AS_XX>
                  <AS_YY>10</AS_YY>
                </Naming>
                <Enable>true</Enable>
              </ConfederationPeerAS>
            </ConfederationPeerASTable>
          </Global>
        </DefaultVRF>
      </FourByteAS>
    </AS>
  </BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</Response>

```

Native Data Operation Examples

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

- [Set Configuration Data Request: Example, page 4-60](#)
- [Get Request: Example, page 4-62](#)
- [Get Request of Nonexistent Data: Example, page 4-63](#)
- [Delete Request: Example, page 4-65](#)
- [GetDataSpaceInfo Request Example, page 4-66](#)

Set Configuration Data Request: Example

This 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 these CLI commands:

```
router bgp 3
  address-family ipv4 unicast!
  address-family ipv4 multicast!
  neighbor 10.0.101.6
    remote-as 6
    ebgp-multihop 255
  address-family ipv4 unicast
    orf route-policy BGP_pass all
      capability orf prefix both
    !
  address-family ipv4 multicast
    orf route-policy BGP_pass all
  !
!
```

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>
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
          <BGPRunning>true</BGPRunning>
          <DefaultVRF>
            <Global>
              <GlobalAFTable>
                <GlobalAF>
                  <Naming>
                    <AFName>IPv4Unicast</AFName>
                  </Naming>
                  <Enable>true</Enable>
                </GlobalAF>
              </GlobalAFTable>
            </Global>
          </DefaultVRF>
        </BGP>
      </Configuration>
    </Set>
  </Request>
</Request>
```

```

    <GlobalAF>
      <Naming>
        <AFName>IPv4Multicast</AFName>
      </Naming>
      <Enable>>true</Enable>
    </GlobalAF>
  </GlobalAFTable>
</Global>
<BGPEntity>
  <NeighborTable>
    <Neighbor>
      <Naming>
        <NeighborAddress>
          <IPV4Address>10.0.101.6</IPV4Address>
        </NeighborAddress>
      </Naming>
      <RemoteAS>
        <AS_XX>0</AS_XX>
        <AS_YY>6</AS_YY>
      </RemoteAS>
      <EBGPMultihop>
        <MaxHopCount>255</MaxHopCount>
        <MPLSDeactivation> false </MPLSDeactivation>
      </EBGPMultihop>
      <NeighborAFTable>
        <NeighborAF>
          <Naming>
            <AFName>IPv4Unicast</AFName>
          </Naming>
          <Activate>>true</Activate>
          <PrefixORFPolicy>BGP_pass_all</PrefixORFPolicy>
          <AdvertiseORF> Both </AdvertiseORF>
        </NeighborAF>
        <NeighborAF>
          <Naming>
            <AFName>IPv4Multicast</AFName>
          </Naming>
          <Activate>>true</Activate>
          <PrefixORFPolicy>BGP_pass_all</PrefixORFPolicy>
        </NeighborAF>
      </NeighborAFTable>
    </Neighbor>
  </NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Set>
<Commit/>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
  <Commit CommitID="1000000029"/>
  <ResultSummary ErrorCount="0"/>
</Response>

```

Get Request: Example

This 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).

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>
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</as>
            </Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.6</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.6</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>
```



```

</Naming>
<RemoteAS>
  <AS XX>0</AS XX>
  <AS YY>6</AS YY>
</RemoteAS>
<EBGPMultihop>
  <MaxHopCount>255</MaxHopCount>
  <MPLSDeactivation>>false</MPLSDeactivation>
</EBGPMultihop>
<NeighborAFTable>
  <NeighborAF>
    <Naming>
      <AFName>IPv4Unicast</AFName>
    </Naming>
    <Activate>>true</Activate>
  <PrefixORFPolicy>BGP_pass_all</PrefixORFPolicy>
  <AdvertiseORF>Both</AdvertiseORF>
</NeighborAF>
<NeighborAF>
  <Naming>
    <AFName>IPv4Multicast</AFName>
  </Naming>
  <Activate>>true</Activate>
  <PrefixORFPolicy>BGP_pass_all</PrefixORFPolicy>
</NeighborAF>
</NeighborAFTable>
</Neighbor>
</NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</Response>

```

Get Request of Nonexistent Data: Example

This example shows a native data request to get the configuration values for a particular BGP neighbor; this is 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
        <FourByteAS>

```

```

    <Naming>
      <AS>3</AS>
    </Naming>
  <DefaultVRF>
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <NeighborAddress>
              <IPV4Address>10.0.101.99</IPV4Address>
            </NeighborAddress>
          </Naming>
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get ItemNotFound="true">
    <Configuration>
      <BGP MajorVersion="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
        <FourByteAS>
          <Naming>
            <AS>3</AS>
          </Naming>
        <DefaultVRF>
          <BGPEntity>
            <NeighborTable>
              <Neighbor NotFound="true">
                <Naming>
                  <NeighborAddress>
                    <IPV4Address>10.0.101.99</IPV4Address>
                  </NeighborAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </DefaultVRF>
      </FourByteAS>
    </AS>
  </BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0" ItemNotFound="true"/>
</Response>

```

Delete Request: Example

This 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 this 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 these 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.6</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Delete>
  <Commit/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Delete>
    <Configuration/>
  </Delete>
  <Commit CommitID="1000000030"/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

GetDataSpaceInfo Request Example

This example shows a `<GetDataSpaceInfo>` operation used to retrieve the native data branch names dynamically. This is useful, for example, for writing a client application that can issue a `<GetVersionInfo>` operation without having to hardcode the branch names. The `<GetDataSpaceInfo>` operation can be invoked instead to retrieve the branch names. The returned branch names can then be included in a subsequent `<GetVersionInfo>` request.

Sample XML Client Request to Retrieve Native Data

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

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1"
  MinorVersion="0">
  <GetDataSpaceInfo>
    <Configuration/>
    <Operational/>
    <Action/>
    <AdminOperational/>
    <AdminAction/>
  </GetDataSpaceInfo>
  <ResultSummary ErrorCount="0" />
</Response>
```



CHAPTER 5

Cisco XML and Native Data Access Techniques

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

Available Set of Native Data Access Techniques

The available native data access techniques are:

- Request all data in the configuration hierarchy. See the [“XML Request for All Configuration Data” section on page 5-68](#).
- Request all configuration data for a component. See the [“XML Request for All Configuration Data per Component” section on page 5-68](#).
- Request all data within a container. See the [“XML Request for Specific Data Items” section on page 5-71](#).
- Combine object class hierarchies within a request. See the [“XML Request with Combined Object Class Hierarchies” section on page 5-72](#).
- Use wildcards in order to apply an operation to a set of entries within a table (Match attribute). See the [“XML Request Using Wildcarding \(Match Attribute\)” section on page 5-75](#).
- Repeat naming information in order to apply an operation to multiple instances of an object. See the [“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 the [“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 the [“Limiting the Number of Table Entries Returned \(Count Attribute\)” section on page 5-83](#).
- Use custom filters to filter table entries (Filter element). See the [“Custom Filtering \(Filter Element\)” section on page 5-85](#).
- Use the Mode attribute. See the [“XML Request Using the Mode Attribute” section on page 5-86](#).

The actual data returned in a <Get> request depends on the value of the Source attribute.



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.

This 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>
  <ResultSummary ErrorCount="0" />
</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 this 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="24" 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>
      <BGP MajorVersion="24" MinorVersion="0">
        .
        .
        .
        response data goes here
        .
        .
        .
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>
```

XML Request for All Data Within a Container

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

This 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
```

```

        <NeighborAddress>
          <IPV4Address>10.0.101.6</IPV4Address>
        </NeighborAddress>
      </Naming>
    </Neighbor>
  </NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <NeighborAddress>
                      <IPV4Address>10.0.101.6</IPV4Address>
                    </NeighborAddress>
                  </Naming>
                  <RemoteAS>
                    <AS XX>0</AS XX>
                    <AS YY>6</AS YY>
                  </RemoteAS>
                  <EBGPMultihop>
                    <MaxHopCount>255</MaxHopCount>
                    <MPLSDeactivation>>false</MPLSDeactivation>
                  </EBGPMultihop>
                  <NeighborAFTable>
                    <NeighborAF>
                      <Naming>
                        <AFName>IPv4Unicast</AFName>
                      </Naming>
                      <Activate>>true</Activate>
                      <PrefixORFPolicy>oBGP_pass_all</PrefixORFPolicy>
                      <AdvertiseORF>Both</AdvertiseORF>
                    </NeighborAF>
                    <NeighborAF>
                      <Naming>
                        <AFName>IPv4Multicast</AFName>
                      </Naming>
                      <Activate>>true</Activate>
                      <PrefixORFPolicy>BGP_pass_all</PrefixORFPolicy>
                    </NeighborAF>
                  </NeighborAFTable>
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Response>

```



```

        </NeighborTable>
      </BGPEntity>
    </DefaultVRF>
  </FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</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.

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

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

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="35" MinorVersion="2">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.6</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                    <RemoteAS/>
                    <EBGPMultiHop/>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </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="35" MinorVersion="2">
        <AS>
          <Naming>

```

```

    <AS>0</AS>
  </Naming>
<FourByteAS>
  <Naming>
    <AS>3</AS>
  </Naming>
<DefaultVRF>
  <BGPEntity>
    <NeighborTable>
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <IPv4Address>10.0.101.6</IPv4Address>
          </NeighborAddress>
        </Naming>
      <EBGPMultihop>
        <MaxHopCount>255</MaxHopCount>
      </EBGPMultihop>
    </Neighbor>
  </NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0" />
</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.

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, these 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 the “[XML Request for All Data Within a Container](#)” section on page 5-69.
- An operation to retrieve the list of entries in a table. For more information, See the “[XML Request Using Operation Scope \(Content Attribute\)](#)” section on page 5-82.
- An operation which includes a wildcard. For more information, See the “[XML Request Using Wildcarding \(Match Attribute\)](#)” section on page 5-75.

If an attempt is made to make such an operation followed by another operation within the same request, this error is 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.

These two examples illustrate two different object class hierarchies that retrieve the same data: the value of the leaf object <RemoteAS> and <EBGPMultihop> 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>
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming>
              <AS>3</AS>
            </Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.6</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                    <!-- Gets the following two leaf objects for this neighbor -->
                    <RemoteAS/>
                    <EBGPMultihopMaxHopCount/>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <Get>
    <Configuration>
      <BGP>
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming>AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.7</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

```

    </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>
  <ResultSummary ErrorCount="0" />
</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="35" MinorVersion="2">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPv4Address>10.0.101.6</IPv4Address>
                      </NeighborAddress>
                    </Naming>
                    <!-- Gets the following two leaf objects for this neighbor -->
                    <RemoteAS/>
                    <EBGPMultihop/>
                  </Neighbor>
                </Neighbor>
              </Neighbor>
            </Neighbor>
          </Neighbor>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>

```

```

        <NeighborAddress>
          <!-- Gets all configuration data for this neighbor -->
          <IPV4Address>10.0.101.7</IPV4Address>
        </NeighborAddress>
      </Naming>
    </Neighbor>
  </NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</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 both
      neighbors
      .
      .
      .
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</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.

If no match is found, the response message contains MatchFoundBelow="false" in the <Get> class, and MatchFound="false" in the class that specified Match="*" and no match found. These attributes are not added (with a value of true) in the response if a match is found.



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.

This 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="35" MinorVersion="2">
        <AS>

```

```

<Naming><AS>0</AS></Naming>
<FourByteAS>
  <Naming><AS>3</AS></Naming>
  <DefaultVRF>
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <NeighborAddress Match="*" />
          </Naming>
          <RemoteAS />
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </DefaultVRF>
</FourByteAS>
</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="35" MinorVersion="2">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.1</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                    <RemoteAS>1</RemoteAS>
                  </Neighbor>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.2</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                    <RemoteAS>2</RemoteAS>
                  </Neighbor>
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.3</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                    <RemoteAS>3</RemoteAS>
                  </Neighbor>
                <...
                  <i>data for more neighbors
                  returned here</i>
                <...

```

```

        </NeighborTable>
      </BGPEntity>
    </DefaultVRF>
  </FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</Response>

```

This example shows the response message when there is no match found for the request with wildcarding:

Sample XML Client Request for No Match Found with Wildcarding

```

<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <BGP MajorVersion="35" MinorVersion="2">
        <AS>
          <Naming><AS>3</AS>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <NeighborAddress Match="*" />
                  </Naming>
                  <RemoteAS />
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </FourByteAS>
      </AS>
    </BGP>
  </Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get VersionMismatchExistsBelow="true" MatchFoundBelow="false" \ ItemNotFound="true">
    <Configuration>
      <BGP MajorVersion="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>3</AS>
          </Naming>
        <FourByteAS>
          <Naming>
            <AS>3</AS>
          </Naming>
        <DefaultVRF>
          <BGPEntity>

```

```

    <NeighborTable>
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <Address Match="*" MatchFound="false" />
          </NeighborAddress>
        </Naming>
        <RemoteAS NotFound="true" />
      </Neighbor>
    </NeighborTable>
  </BGPEntity>
</DefaultVrF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0" ItemNotFound="true" />
</Response>

```

Regular expression matching of naming information is provided by means of the Match attribute. Match="**<regular expression>**" can be used on any Naming attribute within a <Get> operation to specify a filtering criteria to filter table entries.

These rules apply to the filtering criteria:

- The character, '*', is treated same as the '.' character. (matches everything)
- Meta character '^' (beginning of line) and '\$' (end of line) are always attached to the regular expression string specified by 'Match' attribute.
- A regular expression string without any meta characters is treated as an exact match.

Sample Request of the Configured ACL Entries That End With 'SAA':

```

<Get>
  <Configuration>
    <IPV4_ACLAndPrefixList>
      <AccessListTable>
        <AccessList>
          <Naming>
            <AccessListName Match=".*SAA" />
          </Naming>
        </AccessList>
      </AccessListTable>
    </IPV4_ACLAndPrefixList>
  </Configuration>
</Get>

```

ACL entries that match this request: TCLSAA, 100SAA, SAA

ACL entries that do NOT match this request: TCLSAA1

Sample Request That Returns all of the Configured GigabitEthernet Ports in Slot 5:

```

<Get>
  <Configuration>
    <Configuration>
      <InterfaceConfigurationTable>
        <InterfaceConfiguration>
          <Naming>
            <Active>act</Active>
            <InterfaceName Match="GigabitEthernet0/5/[0-9]+/[0-9]+" />
          </Naming>
        </InterfaceConfiguration>
      </InterfaceConfigurationTable>
    </Configuration>
  </Configuration>
</Get>

```



```

    </InterfaceConfigurationTable>
  </Configuration>
</Get>

```

Interface names that match this request: GigabitEthernet0/5/0/0, GigabitEthernet0/5/0/1, and so forth.

Interface names that do not match this request: GigabitEthernet0/4/0/0

Sample Request That Returns the Configured Loopback Interfaces Between Loopback100 and Loopback199:

```

<Get>
  <Configuration>
    <Configuration>
      <InterfaceConfigurationTable>
        <InterfaceConfiguration>
          <Naming>
            <Active>act</Active>
            <InterfaceName Match="Loopback1[0-9][0-9]" />
          </Naming>
        </InterfaceConfiguration>
      </InterfaceConfigurationTable>
    </Configuration>
  </Get>

```

Interface names that match this request: Loopback100,...,Loopback199

Interface names that do not match this request: Loopback1000, Loopback1990

Sample Request That Returns Only Loopback1 (if it is configured):

```

<Get>
  <Configuration>
    <Configuration>
      <InterfaceConfigurationTable>
        <InterfaceConfiguration>
          <Naming>
            <Active>act</Active>
            <InterfaceName Match="Loopback1" />
          </Naming>
        </InterfaceConfiguration>
      </InterfaceConfigurationTable>
    </Configuration>
  </Get>

```

Interface names that match this request: Loopback1

Interface names that do not match this request: Loopback10, Loopback100, and so forth

The request above, thus, is equivalent to this request:

```

<Get>
  <Configuration>
    <Configuration>
      <InterfaceConfigurationTable>
        <InterfaceConfiguration>
          <Naming>
            <Active>act</Active>
            <InterfaceName>Loopback1</InterfaceName>
          </Naming>
        </InterfaceConfiguration>
      </InterfaceConfigurationTable>
    </Configuration>
  </Get>

```

Limitation: Regular expression matching can only be specified in the first table of an XML request.

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.

This 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>
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
        <FourByteAS>
          <Naming><AS>3</AS></Naming>
        <DefaultVRF>
          <BGPEntity>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <NeighborAddress>
                    <IPV4Address>10.0.101.1</IPV4Address>
                  </NeighborAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <NeighborAddress>
                    <IPV4Address>10.0.101.6</IPV4Address>
                  </NeighborAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
            <NeighborTable>
              <Neighbor>
                <Naming>
                  <NeighborAddress>
                    <IPV4Address>10.0.101.8</IPV4Address>
                  </NeighborAddress>
                </Naming>
              </Neighbor>
            </NeighborTable>
          </BGPEntity>
        </DefaultVRF>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

```

        </Neighbor>
      </NeighborTable>
    </BGPEntity>
  </DefaultVRF>
</FourByteAS>
</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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          <DefaultVRF>
            <BGPEntity>
              <NeighborTable>
                <Neighbor>
                  <Naming>
                    <NeighborAddress>
                      <IPV4Address>10.0.101.1</IPV4Address>
                    </NeighborAddress>
                  </Naming>
                  ...
                  data returned for 1st neighbor
                  ...
                </Neighbor>
                <Neighbor>
                  <Naming>
                    <NeighborAddress>
                      <IPV4Address>10.0.101.6</IPV4Address>
                    </NeighborAddress>
                  </Naming>
                  ...
                  data returned for 2nd neighbor
                  ...
                </Neighbor>
                <Neighbor>
                  <Naming>
                    <NeighborAddress>
                      <IPV4Address>10.0.101.6</IPV4Address>
                    </NeighborAddress>
                  </Naming>
                  ...
                  data returned for 3rd neighbor
                  ...
                </Neighbor>
              </NeighborTable>
            </BGPEntity>
          </DefaultVRF>
        </FourByteAS>
      </AS>
    </BGP>
  </Configuration>

```

```

</Get>
<ResultSummary ErrorCount="0"/>
</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	Used to get all leaf items and their values. All is the default when the Content attribute is not specified on a table element.
Entries	Used 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. Also, note that the Content and Count attributes can be used together on the same table element.

This 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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable Content="Entries"/>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>

```

```

<FourByteAS>
  <Naming><AS>3</AS></Naming>
  <DefaultVRF>
  <BGPEntity>
    <NeighborTable Content="Entries">
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <IPv4Address>10.0.101.1</IPv4Address>
          </NeighborAddress>
        </Naming>
      </Neighbor>
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <IPv4Address>10.0.101.2</IPv4Address>
          </NeighborAddress>
        </Naming>
      </Neighbor>
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <IPv4Address>10.0.101.3</IPv4Address>
          </NeighborAddress>
        </Naming>
      </Neighbor>
      <Neighbor>
        <Naming>
          <NeighborAddress>
            <IPv4Address>10.0.101.4</IPv4Address>
          </NeighborAddress>
        </Naming>
      </Neighbor>
      ...
      more neighbors returned here
      ...
    </NeighborTable>
  </BGPEntity>
</DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</Request>

```

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.

This 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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable Count="5">
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>10.0.101.1</IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                </NeighborTable>
              </BGPEntity>
            </DefaultVRF>
          </FourByteAS>
        </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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <BGPEntity>
                <NeighborTable Count="5">
                  <Neighbor>
                    <Naming>
                      <NeighborAddress>
                        <IPV4Address>
                          10.0.101.1
                        </IPV4Address>
                      </NeighborAddress>
                    </Naming>
                  </Neighbor>
                <Neighbor>
                  <Naming>
                    <NeighborAddress>
```

```

        <IPV4Address>
            10.0.101.2
        </IPV4Address>
    </NeighborAddress>
</Naming>
</Neighbor>
...
    data returned for remaining
    neighbors here
...
</NeighborTable>
</BGPEntity>
</DefaultVRF>
</FourByteAS>
</AS>
</BGP>
</Configuration>
</Get>
<ResultSummary ErrorCount="0"/>
</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.

In this 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>
        <Active>
          <VRFTable>
            <VRF>
              <Naming>
                <VRFName>one</VRFName>
              </Naming>
              <NeighborTable>
                <Filter>
                  <BGP_ASFilter>
                    <AS>6</AS>
                  </BGP_ASFilter>
                </Filter>
              </NeighborTable>
            </VRF>
          </VRFTable>
        </Active>
      </BGP>
    </Operational>
  </Get>
</Request>

```

```

</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="23" MinorVersion="7">
        <Active>
          <VRFTable>
            <VRF>
              <Naming>
                <VRFName>one</VRFName>
              </Naming>
              <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>
            </VRF>
          </VRFTable>
        </Active>
      </BGP>
    </Operational>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>

```

XML Request Using the Mode Attribute

The client application modifies the target configuration as needed using the <Delete> and <Set> operations. The XML interface supports the combining of several operations into a single request. When multiple configuring operations are specified in a single request, they are performed on a “best effort” basis by default. For example, in a case where configuring operations 1 through 3 are in the request and even if operation 2 fails, operation 3 is attempted and operation 1 result remains in the target configuration.

To perform the request on an atomic basis, use the Mode attribute with the value Atomic in the <Request>. If any errors occur, the target configuration is cleared and the errors are returned to the client application.

Sample XML Client Request with the Attribute Mode="Atomic"

```
<?xml version='1.0' encoding='UTF-8'?>
<Request Version="1.0" Mode="Atomic">
  <Set>
    <Configuration>
      <SNMP>
        <Timeouts>
          <Subagent> 20 </Subagent>
        </Timeouts>
      </SNMP>
    </Configuration>
  </Set>
  <Commit/>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
  <Commit CommitID="1000000441"/>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Sample XML Client Request with an Invalid Set Operation (Best-Effort)

```
<?xml version="1.0" encoding="UTF-8"?>
<Request Version="1.0">
  <Set>
    <Configuration>
      <SNMP>
        <Timeouts>
          <Subagent> 20 </Subagent>
        </Timeouts>
      </SNMP>
    </Configuration>
  </Set>
  <Set>
    <Configuration>
      <SNMP>
        <System>
          <Contact> </Contact> <--- This is an invalid XML set operation
        </System>
      </SNMP>
    </Configuration>
  </Set>
  <Commit/>
</Request>
```

Sample XML Response from the Router

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

```

    <SNMP MajorVersion="7" MinorVersion="3">
      <System>
        <Contact ErrorCode="0x4368b000" ErrorMessage="&apos;XMLMDA&apos; detected
the &apos;warning&apos; condition &apos;The XML request does not conform to the schema.
The XML below the element on which this error appears is missing a required piece of data.
Please check the request against the schema.&apos;"/>
      </System>
    </SNMP>
  </Configuration>
</Set>
<Commit CommitID="1000000443"/>
<ResultSummary ErrorCount="1"/>
</Response>

```

**Note**

This request is performed on a best effort basis. The SNMP timeout configuration has no error and is committed.

Sample XML Request and Response of Commit Change for ForCommitID="1000000443"

```

<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Get>
    <Configuration Source="CommitChanges" ForCommitID="1000000443"/>
  </Get>
</Request>

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration Source="CommitChanges" ForCommitID="1000000443"
OperationType="Set">
      <SNMP MajorVersion="7" MinorVersion="3">
        <Timeouts>
          <Subagent>
            20
          </Subagent>
        </Timeouts>
      </SNMP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>

```

Sample XML Client Request with the Attribute Mode="Atomic" and with an Invalid Set Operation

```

<?xml version="1.0" encoding="UTF-8"?>
<Request Version="1.0" Mode="Atomic">
  <Set>
    <Configuration>
      <SNMP>
        <Timeouts>
          <Subagent> 20 </Subagent>
        </Timeouts>
      </SNMP>
    </Configuration>
  </Set>
  <Set>
    <Configuration>
      <SNMP>
        <System>

```

```

        <Contact> </Contact>    <--- This is an invalid XML set operation
    </System>
  </SNMP>
</Configuration>
</Set>
  <Commit/>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set>
    <Configuration/>
  </Set>
  <Set ErrorCode="0x43679000" ErrorMessage="&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>
      <SNMP MajorVersion="7" MinorVersion="3">
        <System>
          <Contact ErrorCode="0x4368b000" ErrorMessage="&apos;XMLMDA&apos; detected
the &apos;warning&apos; condition &apos;The XML request does not conform to the schema.
The XML below the element on which this error appears is missing a required piece of data.
Please check the request against the schema.&apos; "/>
          </System>
        </SNMP>
      </Configuration>
    </Set>
    <Commit ErrorCode="0x41864e00" ErrorMessage="&apos;CfgMgr&apos; detected the
&apos;warning&apos; condition &apos;The target configuration buffer is empty.&apos; "/>
    <ResultSummary ErrorCount="1"/>
  </Response>

```



Note

The target configuration buffer is cleared and no configuration is committed.



CHAPTER 6

Cisco XML and Encapsulated CLI Operations

XML interface for the router provides support for XML encapsulated 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, <Configuration> tags, and <EXEC> 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.

This 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
    commit
</Configuration>
<Exec>
    sh config commit changes last 1
</Exec>
</CLI>
</Request>

```

Sample XML Response from the Router

```

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

```

CLI Command Limitations

The CLI commands, which are supported through XML, are limited to CLI configuration commands and EXEC mode **show** commands (and responses) that are wrapped in <CLI> tags.

These commands and conditions are not supported:

- The **do** configuration mode command.
- EXEC mode commands other than show commands except for these items:
 - **show history**
 - **show user**
 - **show users**
 - **show terminal**
- Administration EXEC mode 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.
- Only one XML <CLI> request can be issued at a time across all client sessions on the router.



CHAPTER 7

Cisco XML and Large Data Retrieval

XML for the router supports the retrieval of large XML responses in blocks (for example, chunks or sections).

These sections provide information about large data retrieval:

- [Iterators, page 7-93](#)
- [Throttling, page 7-98](#)
- [Streaming, page 7-99](#)

Iterators

When a client application makes a request, the resulting response data size is checked to determine whether it is larger than a predetermined block size.

If the response data is not larger than the predetermined block size, the complete data is returned in a normal response.

If the response data is larger than the block size, the first set of data is returned according to the block size along with a decremented iterator ID included as the value of the `IteratorID` attribute. The client must then send `<GetNext>` requests including the iterator ID until all data is retrieved. The client application knows that all data is retrieved when it receives a response that does not contain an `IteratorID` attribute.

Usage Guidelines

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

- The block size is a configurable value specific to each transport mechanism on the router; that is, the XML agent for the dedicated TCP connection and Secure Shell (SSH), Telnet, or Secure Sockets Layer (SSL) dedicated TCP connection.

Use this command to configure the iteration size:

```
xml agent [tty | ssl] iteration on size <1-100000>
```

Specify the iteration size in KB. The default is 48 KB.



Note The **iteration** command includes the option to turn off the XML response iterator. However, we do not recommend turning off the iterator because of the large memory usage that occurs temporarily.

- 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 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 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, these limitations are placed on the number of allowed iterators:

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

If a <Get> request is issued that results in an iterated response, it is counted as one 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 the associated data, but during this process only one iterator is being used.

Also, an iterator is considered to be in use until all of the response data associated with that iterator (the original <Get> request) is retrieved or the iterator is terminated with the Abort attribute.

Examples Using Iterators to Retrieve Data

This 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="35" MinorVersion="2">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          <DefaultVRF/>
        </FourByteAS>
      </AS>
    </BGP>
  </Get>
</Request>
```



```

    </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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              1st block of data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</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>

```

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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              2nd block of data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              3rd block of data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0" />
</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>
```

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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              Final block of data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0" />
</Response>
```

Large Response Division

The default behavior for large response division is that large responses are divided based on the requested block size.

To specify a different basis for the division, use the `IterateAtFirstTableGet` attribute in the `<Get>` tag.

Sample XML Request with attribute `IterateAtFirstTable`

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Get IterateAtFirstTable="true">
    <Operational>
      <BGP>
        <Active>
          <DefaultVRF>
            <AFTable>
              <AF>
                <Naming>
                  <AFName Match="*" />
                </Naming>
              <PathTable>
                <Path>
                  <Naming>
                    <RD Match="*" />
                    <Network Match="*" />
                    <NeighborAddress Match="*" />
                    <RouteType Match="*" />
                    <SourceRD Match="*" />
                  </Naming>
                </Path>
              </PathTable>
            </AF>
          </AFTable>
        </DefaultVRF>
      </Active>
    </BGP>
  </Operational>
</Get>
</Request>
```

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.

This 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
        <FourByteAS>
```

```

        <Naming><AS>3</AS></Naming>
        <DefaultVRF/>
        </FourByteAS>
    </AS>
</BGP>
</Configuration>
</Get>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="2">
  <Get>
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              1st block of data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</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>

```

Throttling

XML response data could be large resulting in high CPU utilization or high memory usage when constructing the XML response. Throttling mechanisms in the XML agent provide a means for external users or an NMS to control the impact to the system.

CPU Throttle Mechanism

The CPU throttle mechanism in the XML agent controls the number of tags to process per second. The higher the number of tags that are specified, the higher the CPU utilization and faster response. The lower number of tags means less CPU utilization and slower response.

To configure the number of tags, use this command:

```
xml agent [tty | ssl] throttle process-rate <1000-30000>
```

Memory Throttle Mechanism

The memory throttle mechanism in the XML agent controls the maximum XML response size in MB. If this size is exceeded, this error message is returned in the XML response.

```
> XML> <?xml version="1.0" encoding="UTF-8"?>
> <Response MajorVersion="1" MinorVersion="0"><Get ErrorCode="0xa367a600"
ErrorMsg="&apos;XML Service Library&apos; detected the &apos;fatal&apos; condition
&apos;The throttle on the memory usage has been reached. Please optimize the request to
query smaller data.&apos;"/></Response>
```

To configure the size of the memory usage per session, use this command:

```
xml agent [tty | ssl] throttle memory <100-600>
```

The default is 300 MB.

Streaming

As the XML agent retrieves the data from the source, the output of a response is streamed. This process is similar to iterators, but the XML client does not run the GetNext IteratorID to handle large response data size.

Usage Guidelines

Use these guidelines when streaming is used by the client application:

- Iteration must be off.

```
xml agent [tty | ssl] iteration off
```

- The sub-response block size is a configurable value specific to each transport mechanisms on the router: the XML agent for the dedicated TCP connection and Secure Shell (SSH), Telnet, or Secure Sockets Layer (SSL) dedicated TCP connection.

Use this command to configure the streaming size. Specify the streaming size in KB. The default is 48 KB.

```
xml agent [tty | ssl] streaming on size <1-100000>
```




CHAPTER 8

Cisco XML Security

Specific security privileges are required for a client application requesting information from the router.

This chapter contains these sections:

- [Authentication, page 8-101](#)
- [Authorization, page 8-101](#)
- [Retrieving Task Permissions, page 8-102](#)
- [Task Privileges, page 8-102](#)
- [Task Names, page 8-103](#)
- [Authorization Failure, page 8-104](#)
- [Management Plane Protection, page 8-104](#)
- [VRF, page 8-105](#)
- [Access Control List, page 8-105](#)

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 or assigned for any objects accessed in the operation:

- <Get> 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.

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 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.

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

Sample XML Request to Retrieve AAA Configuration Information

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get>
    <Configuration>
      <AAA MajorVersion="2" 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="2" MinorVersion="0">
        .
        .
        .
        AAA configuration returned here
        .
        .
      </AAA>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Task Privileges

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

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

The “configuration services” operations through the 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 this annotation:

```
<xsd:appinfo>
  <ObjectType>Container</ObjectType>
  <MajorVersion>18</MajorVersion>
  <MinorVersion>0</MinorVersion>
  <TaskIDInfo TaskGrouping="Single">
    <TaskName>bgp</TaskName>
  </TaskIDInfo>
  <Parents>
    <Parent>
      <Schema>native_data_operations</Schema>
      <Name>Configuration</Name>
    </Parent>
  </Parents>
</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 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.

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.

Management Plane Protection

Management Plane Protection (MPP) provides a mechanism for securing management traffic on the router. Without MPP, a management service’s traffic can come through any interface with a network address, which could be a security risk.

MPP is effective when XML is configured.

Inband Traffic

To configure the MPP for inband traffic, use the command in this example:

```
RP/0/0/CPU0:router(config)#control-plane management-plane inband interface [interface
type] allow [protocol|all]
```

where the **protocol** is XML.

```
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#Ethernet 0/0/0/0 allow XML ?
peer Configure peer address on this interface
<cr>
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#Ethernet 0/0/0/0 allow XML peer ?
address Configure peer address on this interface
<cr>
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#Ethernet 0/0/0/0 allow XML peer address ?
ipv4 Configure peer IPv4 address on this interface
ipv6 Configure peer IPv6 address on this interface
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#Ethernet 0/0/0/0 allow XML peer address
```

Out-of-Band Traffic

To configure the MPP for out-of-band traffic, use the command in this example:

```
RP/0/0/CPU0:router(config)#control-plane management-plane out-of-band interface
[interface type] allow [protocol|all]
```

where the **protocol** is XML.

```
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#GigabitEthernet 0/0/0/1 allow XML ?
peer Configure peer address on this interface
<cr>
RP/0/RSP0/CPU0:PE44_ASR-9010(config)#GigabitEthernet 0/0/0/1 allow XML peer ?
address Configure peer address on this interface
<cr>
```

```
RP/0/RSP0/CPU0:PE44_ASR-9010(config)# XML peer address ?
  ipv4  Configure peer IPv4 address on this interface
  ipv6  Configure peer IPv6 address on this interface
RP/0/RSP0/CPU0:PE44_ASR-9010(config)# XML peer address
```

VRF

XML agents can be configured to virtual route forwarding (VRF) aware.

- To configure the dedicated agent [ssl] to receive or send messages through VRF, use this command:

```
RP/0/0/CPU0:router(config)#xml agent [ssl] vrf <vrf name>
```

- To configure the dedicated [ssl] agent NOT to receive or send messages through the default VRF, use this command:

```
RP/0/0/CPU0:Router(config)#xml agent [ssl] vrf default shutdown
```

Access Control List

To configure an access control list (ACL) for XML agents, use this command:

```
RP/0/0/CPU0:router(config)#xml agent [ssl] vrf <vrf name> access-list <access-list name>
```

IPv6 Access List Example

```
xml agent [ssl]
  vrf <vrf name>
    ipv6 access-list <ipv6 access-list name>
```

IPv4 and IPv6 Access Lists Example

```
xml agent [ssl]
  vrf <vrf name>
    ipv4 access-list <ipv4 access-list name>
    ipv6 access-list <ipv6 access-list name>
  !
!
```



Note This method to configure an IPv4 access-list is still supported (for backward compatibility) but hidden from CLI help.

```
xml agent [ssl]
  vrf <vrf name>
    access-list <ipv4 access-list name>
  !
!
```




CHAPTER 9

Cisco XML Schema Versioning

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

Major and minor version numbers are included on the <Request> and <Response> elements to indicate the overall 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 these sections:

- [Major and Minor Version Numbers, page 9-107](#)
- [Run-Time Use of Version Information, page 9-108](#)
- [Retrieving Version Information, page 9-113](#)
- [Retrieving Schema Detail, page 9-115](#)

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.

This 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>BGP Configuration Commands</xsd:documentation>
    <xsd:appinfo>
      <ObjectType>Container</ObjectType>
      <MajorVersion>24</MajorVersion>
      <MinorVersion>0</MinorVersion>
      <TaskIdInfo TaskGrouping="Single">
```

```

        <TaskName>bgp</TaskName>
    </TaskIdInfo>
    <Parents>
        <Parent>
            <Schema>native_data_operations</Schema>
            <Name>Configuration</Name>
        </Parent>
    </Parents>
</xsd:appinfo>
</xsd:annotation>
    .
    .
    .
    <xsd:attributeGroup ref="VersionAttributeGroup"/>
    .
    .
    ..
</xsd:complexType>

```

The attribute group `VersionAttributeGroup` is defined as:

```

<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 major and minor version numbers of the client at the appropriate locations in the XML. These version numbers are compared to the version numbers running on the router.

The behavior of the router, whether the request is accepted or rejected, depends on the value set for the `AllowVersionMismatch` attribute.

All requests are accepted when the `AllowVersionMismatch` attribute is set as `TRUE`. The request is then accepted or rejected based on these rules when the `AllowVersionMismatch` attribute is set as `FALSE`:

- 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.
- The default value is used when the request does not specify the `AllowVersionMismatch` attribute. The default value is currently set as `TRUE`.

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.

Placement of Version Information

This 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="24" MinorVersion="0">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF/>
          </FourByteAS>
        </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="24" MinorVersion="0">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              ...
              data returned here
              ...
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  <Get>
    <ResultSummary ErrorCount="0"/>
</Response>
```

Version Lag with the AllowVersionMismatch Attribute Set as TRUE

The example shows a request and response with a version mismatch. In this case, because the `AllowVersionMismatch` attribute is set as `TRUE`, the request is attempted. This is also the default behavior when `AllowVersionMismatch` attribute is not specified in the request. The router attempts the request and if the request is successful returns a `VersionMismatchExists` attribute at the appropriate point within the response along with a `VersionMismatchExistsBelow` attribute on the `<Get>` operation tag.



Note

The version number, which is returned in the response, is the version running on the router. The versions in this example are hypothetical.

Sample XML Client Request with a Version Mismatch

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get AllowVersionMismatch="true">
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="0">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global/>
            </DefaultVRF>
          </FourByteAS>
        </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 AllowVersionMismatch="true" VersionMismatchExistsBelow="true">
    <Configuration>
      <BGP MajorVersion="24"
        MinorVersion="1">
        VersionMismatchExists="true">
      <AS>
        <Naming><AS>0</AS></Naming>
        <FourByteAS>
          <Naming><AS>3</AS></Naming>
          <DefaultVRF>
            <Global>
              ...
              data returned here
              ...
            </Global>
          </DefaultVRF>
        </FourByteAS>
      </AS>
    </BGP>
  </Configuration>
</Get>
<ResultSummary ErrorCount="0" />
</Response>
```


Version Lag with the AllowVersionMismatch Attribute Set as FALSE

The example shows a request and response with a version mismatch, but the request specifies the AllowVersionMismatch attribute as FALSE.

In this case, the client minor version is behind the router, so the request is still attempted, but VersionMismatchExists and VersionMismatchExistsBelow attributes are not returned in the response.



Note

The version number returned in the response is the version number running on the router. The versions in this example are hypothetical.

Sample XML Client Request with the AllowVersionMismatch Attribute Set as False

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get AllowVersionMismatch="false">
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="0">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global/>
            </DefaultVRF>
          </FourByteAS>
        </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 AllowVersionMismatch="false">
    <Configuration>
      <BGP MajorVersion="24"
        MinorVersion="1">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global>
                ...
                data returned here
                ...
              </Global>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0"/>
</Response>
```

Version Creep with the AllowVersionMismatch Attribute Set as TRUE

The example shows a request and response with a version mismatch. In this case, the client is the AllowVersionMismatch attribute and is set as TRUE. The request is attempted.



Note

The version number returned in the response is the version number running on the router. The versions in this example are hypothetical.

Sample XML Request with an AllowVersion Mismatch Attribute Set as TRUE

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get AllowVersionMismatch="true">
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="1">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global/>
            </DefaultVRF>
          </FourByteAS>
        </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 AllowVersionMismatch="true" VersionMismatchExistsBelow="true">
    <Configuration>
      <BGP MajorVersion="24"
        MinorVersion="0">
        VersionMismatchExists="true">
        <AS>
          <Naming><AS>0</AS></Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global>
                ...
                data returned here
                ...
              </Global>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
  <ResultSummary ErrorCount="0" />
</Response>
```

Version Creep with the AllowVersionMismatch Attribute Set as FALSE

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

Sample XML Request with an AllowVersion Mismatch Attribute Set as FALSE

```
<?xml version="1.0" encoding="UTF-8"?>
<Request MajorVersion="1" MinorVersion="0">
  <Get AllowVersionMismatch="false">
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="1">
    </Configuration>
  </Get>
</Request>
```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0" IteratorID="12345678">
  <Get ErrorCode="0x43679000">
    ErrorMessage="&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="24" MinorVersion="0" ErrorCode="0x4368ac00"
      ErrorMessage="&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>
<ResultSummary ErrorCount="0"/>
</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 this 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="18" MinorVersion="0"/>
    </Configuration>
```

```

    </GetVersionInfo>
    <ResultSummary ErrorCount="0" />
</Response>

```

This 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>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetVersionInfo>
    <Configuration>
      <AAA MajorVersion="2" MinorVersion="0">
        <ServerGroups MajorVersion="2" MinorVersion="3">
          <RADIUSServerGroupTable MajorVersion="2" MinorVersion="0"/>
          <TACACSServerGroupTable MajorVersion="5" MinorVersion="1"/>
        </ServerGroups>
        <TaskgroupTable MajorVersion="2" MinorVersion="3"/>
        <UsergroupTable MajorVersion="2" MinorVersion="3"/>
        <UsernameTable MajorVersion="2" MinorVersion="3"/>
        <DefaultTaskgroup MajorVersion="2" MinorVersion="3"/>
        <RADIUS MajorVersion="2" MinorVersion="0"/>
        <TACACS MajorVersion="5" MinorVersion="1"/>
      </AAA>
      . . . .
      <PPP MajorVersion="1" MinorVersion="0">
        <FSM MajorVersion="1" MinorVersion="0"/>
        <Authentication MajorVersion="1" MinorVersion="0"/>
        <CHAP MajorVersion="1" MinorVersion="0"/>
        <MS-CHAP MajorVersion="1" MinorVersion="0"/>
        <PAP MajorVersion="1" MinorVersion="0"/>
      </PPP>
      . . . .
      <QOS MajorVersion="4" MinorVersion="0"/>
      <IntfVRF MajorVersion="1" MinorVersion="1"/>
      <SONET MajorVersion="3" MinorVersion="0"/>
      <BGP MajorVersion="18" MinorVersion="0"/>
      <OSPF MajorVersion="7" MinorVersion="0"/>
      <IPV4_ACLAndPrefixList MajorVersion="5" MinorVersion="0"/>
      <IPV4Network MajorVersion="4" MinorVersion="0"/>
      <IPV4NetworkGlobal MajorVersion="4" MinorVersion="0"/>
      <NTP MajorVersion="2" MinorVersion="1"/>
      <ISIS MajorVersion="15" MinorVersion="0"/>
      <VRFTable MajorVersion="1" MinorVersion="1">
        <VRF><IPARM MajorVersion="3" MinorVersion="0"/>
          <AFI_SAFI_Table>
            <AFI_SAFI>
              <BGP MajorVersion="18" MinorVersion="0"/>
            </AFI_SAFI>
          </AFI_SAFI_Table>
        </VRF>
      </VRFTable>
      . . .
    </Configuration>

```

```

</GetVersionInfo>
<ResultSummary ErrorCount="0"/>
</Response>

```

Retrieving Schema Detail

The SchemaDetail boolean attribute can now be specified on the <GetVersionInfo> operation to instruct the router to return additional schema detail in the response. If the SchemaDetail attribute is specified in the request, each schema entity in the <GetVersionInfo> response contains three additional boolean attributes listed in [Table 9-1](#).

Table 9-1 Content Attributes

Content Attribute	Description
ContainsNaming	Indicates whether or not the schema entity contains naming information.
Gettable	Indicates whether or not <Get> operations are supported for this schema.
Settable	Indicates whether or not <Set> operations are supported for this schema.

This example shows a request and response with the SchemaDetail attribute:

Sample XML Client Request for Schema Detail

```

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

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <GetVersionInfo SchemaDetail="true">
    <Configuration/>
    ...
    <BGP MajorVersion="10" MinorVersion="1" ContainsNaming="false" Gettable="true"
      Settable="true" Supported="true"/>
    <LACP MajorVersion="1" MinorVersion="0" ContainsNaming="false" Gettable="true"
      Settable="true" Supported="true"/>
    <CDP MajorVersion="1" MinorVersion="1" ContainsNaming="false" Gettable="true"
      Settable="true" Supported="true"/>
    <LR MajorVersion="0" MinorVersion="0" ContainsNaming="false" Gettable="true"
      Settable="true" Supported="false"/>
    <InterfaceConfigurationTable MajorVersion="2" MinorVersion="0"
      ContainsNaming="false" Gettable="true" Settable="true" Supported="true">
      <InterfaceConfiguration ContainsNaming="true" Gettable="true" Settable="true">
        <Bundle MajorVersion="1" MinorVersion="0" ContainsNaming="false" Gettable="true"
          Settable="true" Supported="true"/>
        <LACP MajorVersion="1" MinorVersion="0" ContainsNaming="false" Gettable="true"
          Settable="true" Supported="true"/>
      </InterfaceConfiguration>
    </InterfaceConfigurationTable>
  </GetVersionInfo>
</Response>

```

```
        </InterfaceConfiguration>
    </InterfaceConfigurationTable>
    ...
</Configuration>
</GetVersionInfo>
<ResultSummary ErrorCount="0" />
</Response>
```



CHAPTER 10

Alarms

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 Registration IDs 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.

These sections describe the XML used for every operation:

- [Alarm Registration, page 10-117](#)
- [Alarm Deregistration, page 10-118](#)
- [Alarm Notification, page 10-119](#)

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. These filter criteria are listed:

- SourceID
- Category
- Group
- Context
- Code

- Severity
- BiStateOnly

If it succeeds, the response contains a <Register> tag with a RegistrationID attribute. If it fails, the filter tag that caused the error appears with an error message attribute. This 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>CONFIG</Group>
      <Code>DB_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>
  <ResultSummary ErrorCount="0"/>
</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 that 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.

This 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>
```


Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Alarm>
    <Deregister RegistrationID="123"/>
  </Alarm>
  <ResultSummary ErrorCount="0"/>
</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 these fields for the alarm:

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

This example shows the configuration commit alarm notification:

```
<?xml version= "1.0" encoding= "UTF-8"?>
<Notification MajorVersion="1" MinorVersion="0">
  <Alarm RegistrationID="123">
    <SourceID>RP/0/0/CPU0</SourceID>
    <EventID>84</EventID>
    <Timestamp>1077270612</Timestamp>
    <Category>MGBL</Category>
    <Group>CONFIG</Group>
    <Code>DB_COMMIT</Code>
    <Severity>Informational</Severity>
    <State>NotAvailable</State>
    <CorrelationID>0</CorrelationID>
    <AdditionalText>config[65704]: %MGBL-CONFIG-6-DB_COMMIT : Configuration committed
by user &#39;admin&#39;. Use &#39;show commit changes 1000000490&#39; to view
the changes.</AdditionalText>
  </Alarm>
</Notification>
```




Error Reporting in Cisco XML Responses

The 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

Table 11-1 lists the types of potential errors in XML Responses.

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 these sections:

- [Error Attributes, page 11-122](#)
- [Transport Errors, page 11-122](#)
- [XML Parse Errors, page 11-122](#)
- [XML Schema Errors, page 11-123](#)

- [Operation Processing Errors, page 11-125](#)
- [Error Codes and Messages, page 11-126](#)

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 this 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>
```

```

        <DefaultVRF>
          <BGPEntity>
        </DefaultVRF>
      </FourByteAS>
    </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"
    ErrorMessage="&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" ErrorMessage="&apos;XML Infrastructure &apos;
      detected the &apos;fatal&apos; condition &apos;Opening and ending tag does not
      match&apos;"/>
  </Get>
  <ResultSummary ErrorCount="1"/>
</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 this 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">
  <Get>
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global>
                <ExternalRoutes>10</ExternalRoutes>
              </Global>
            </DefaultVRF>
          </FourByteAS>
        </AS>
      </BGP>
    </Configuration>
  </Get>
</Request>

```

```

    </BGP>
  </Configuration>
</Set>
</Request>

```

Sample XML Response from the Router

```

<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x4368a400"
    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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          <DefaultVRF>
            <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;"/>
          </DefaultVRF>
        </AS>
      </BGP>
    </Configuration>
  </Set>
  <ResultSummary ErrorCount="0"/>
</Response>

```

This 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="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
          <DefaultVRF>
            <Global>
              <GracefulRestartTime>6000</GracefulRestartTime>
            </Global>
          </DefaultVRF>
        </FourByteAS>
      </AS>
    </BGP>
  </Configuration>
</Set>
</Request>

```

Sample XML Response from the Router

```
<?xml version="1.0" encoding="UTF-8"?>
<Response MajorVersion="1" MinorVersion="0">
  <Set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ErrorCode="0x4368a800"
    ErrorMessage="XML Service Library detected the warning condition An error was encountered in the XML beneath this operation tag;">
    <Configuration>
      <BGP MajorVersion="24" MinorVersion="0">
        <AS>
          <Naming>
            <AS>0</AS>
          </Naming>
          <FourByteAS>
            <Naming><AS>3</AS></Naming>
            <DefaultVRF>
              <Global>
                <GracefulRestartTime ErrorCode="0x4368a800" ErrorMessage="XMLMDA detected the warning condition 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 child elements) does not conform to the XML schema for its datatype. Please check the request against the schema."/>
              </Global>
            </DefaultVRF>
          </AS>
        </BGP>
      </Configuration>
    </Set>
    <ResultSummary ErrorCount="1"/>
  </Response>
```

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.

This 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" ErrorMessage="XML
  Service Library detected the fatal condition The XML Infrastructure has been provided with an iterator ID which is not recognized. The
  iterator is either invalid or has timed out."/>
```

Error Codes and Messages

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

The error codes (cerrnos) 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 CLI.



CHAPTER 12

Summary of Cisco XML API Configuration Tags

Table 12-1 provides the CLI to XML application programming interface (API) tag mapping for the router target configuration.

Table 12-1 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 ">

Table 12-1 *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 is 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.



CHAPTER 13

XML Transport and Event Notifications

This chapter contains these sections:

- [TTY-Based Transports, page 13-129](#)
- [Dedicated Connection Based Transports, page 13-131](#)
- [SSL Dedicated Connection based Transports, page 13-133](#)

TTY-Based Transports

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

- [Enabling the TTY XML Agent, page 13-129](#)
- [Enabling a Session from a Client, page 13-130](#)
- [Sending XML Requests and Receiving Responses, page 13-130](#)
- [Configuring Idle Session Timeout, page 13-132](#)
- [Ending a Session, page 13-130](#)
- [Errors That Result in No XML Response Being Produced, page 13-131](#)

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 this 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 *Cisco IOS XR System Management Configuration Guide*.

TTY (SSH) agent is telnet based, so IPv6 addressing is supported.

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 total can be started over a dedicated port, TTY, SSH, and Secure Sockets Layer (SSL) dedicated port.

**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.

Configuring Idle Session Timeout

When a session times out, the resource from that session is reclaimed. By default, XML agents do not have an idle session timeout.

To configure the idle session timeout in minutes for the XML agents, use this command:

```
xml agent [tty | ssl] session timeout <1-1440>
```

Ending a Session

If you are using a session interactively from a terminal window, you can close the window. 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.

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 this format:

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

Dedicated Connection Based Transports

These sections describe how to use the dedicated connection-based transports:

- [Enabling the Dedicated XML Agent, page 13-131](#)
- [Enabling a Session from a Client, page 13-132](#)
- [Sending XML Requests and Receiving Responses, page 13-132](#)
- [Configuring Idle Session Timeout, page 13-132](#)
- [Ending a Session, page 13-132](#)
- [Errors That Result in No XML Response Being Produced, page 13-132](#)

Enabling the Dedicated XML Agent

To enable the dedicated agent on the router, which is ready to handle incoming XML sessions over a dedicated TCP port (38751), enter the **xml agent** command, as shown in the following example:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# xml agent
RP/0/RP0/CPU0:router(config)# aaa authorization exec default local
RP/0/RP0/CPU0:router(config)# commit
RP/0/RP0/CPU0:router(config)# exit
```

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

The default addressing protocol for the XML dedicated agent is

- IPv4 enabled
- IPv6 disabled

To configure a dedicated agent to receive and send messages through IPv6 protocol:

```
xml agent ipv6 enable
```

To configure dedicated agent to disable IPv4 protocol

```
xml agent ipv4 disable
```

To receive and send messages only through IPv6 protocol:

```
xml agent ipv4 disable
```

```
xml agent ipv6 enable
```

Enabling a Session from a Client

To enable a session from a remote client, establish a TCP connection with the dedicated port (38751) on the router. When prompted, enter a valid username and password. After you have successfully logged on, the session is in XML mode and is ready to receive XML requests.

A maximum of 50 XML sessions total can be started over dedicated port, TTY, SSH, and SSL dedicated port.

Sending XML Requests and Receiving Responses

To send an XML request, write the request to the established 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.

Configuring Idle Session Timeout

When a session times out, the resource from that session is reclaimed. By default, XML agents do not have an idle session timeout.

To configure the idle session timeout in minutes for the XML agents, use this command:

```
xml agent [tty | ssl] session timeout <1-1440>
```

Ending a Session

If you are using a session interactively from a terminal window, you can close the window. 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.

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 this format:

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

SSL Dedicated Connection based Transports

These sections describe how to use the dedicated connection based transports:

- [Enabling the SSL Dedicated XML Agent, page 13-133](#)
- [Enabling a Session from a Client, page 13-133](#)
- [Sending XML Requests and Receiving Responses, page 13-133](#)
- [Configuring Idle Session Timeout, page 13-134](#)
- [Ending a Session, page 13-134](#)
- [Errors That Result in No XML Response Being Produced, page 13-134](#)

Enabling the SSL Dedicated XML Agent

To enable the SSL dedicated agent on the router, which is ready to handle incoming XML sessions over dedicated TCP port (38752), enter the **xml agent** command, as shown in this example:

```
RP/0/RP0/CPU0:router# configure
RP/0/RP0/CPU0:router(config)# xml agent ssl
RP/0/RP0/CPU0:router(config)# aaa authorization exec default local
RP/0/RP0/CPU0:router(config)# commit
RP/0/RP0/CPU0:router(config)# exit
```



Note

The k9sec package is required to use the SSL agent. The configuration is rejected during a commit when the k9sec package is not active on the system. When the k9sec package is deactivated after configuring the SSL agent, the agent is not available.

The SSL dedicated agent uses IPSec, so IPv6 addressing is supported.

Enabling a Session from a Client

To enable a session from a remote client, establish a TCP connection with the dedicated port (38752) on the router. When prompted, enter a valid username and password. After you have successfully logged on, the session is in XML mode and is ready to receive XML requests.

A maximum of 50 XML sessions can be started over a dedicated port, TTY, SSH, and a SSL dedicated port.

Sending XML Requests and Receiving Responses

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

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>. Asynchronous responses end with </Notification>.

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

Configuring Idle Session Timeout

When a session times out, the resource from that session is reclaimed. By default, XML agents do not have an idle session timeout.

To configure the idle session timeout in minutes for the XML agents, use this command:

```
xml agent [tty | ssl] session timeout <1-1440>
```

Ending a Session

If you are using a session interactively from a terminal window, you can close the window. 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.

Errors That Result in No XML Response Being Produced

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

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




CHAPTER 14

Cisco XML Schemas

This chapter contains information about common XML schemas. The structure and allowable content of the 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 this URL:

<http://www.w3.org/XML/Schema>

This chapter contains these sections:

- [XML Schema Retrieval, page 14-135](#)
- [Common XML Schemas, page 14-136](#)
- [Component XML Schemas, page 14-136](#)

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 this URL to display the Downloads page:

<http://tools.cisco.com/support/downloads/go/Redirect.x?mdfid=268437899>



Note Select Downloads. Only customer or partner viewers can access the Download Software page. Guest users will get an error.

2. Select Cisco IOS XR Software.
3. Select IOS XR XML Schemas.
4. 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.

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. These 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.

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.



CHAPTER 15

Network Configuration Protocol

Network Configuration Protocol (NETCONF) defines an XML-based interface between a network device and a network management system to provide a mechanism to manage, configure, and monitor a network device.

In Cisco IOS-XR, NMS applications use defined XML schemas to manage network devices from multiple vendors. These capabilities are supported from a Cisco IOS XR agent to a client:

- TTY NETCONF session—Logon through telnet and then enter the **netconf** command.
- SSH NETCONF session—Logon through SSH and then enter the **netconf** command.

This example shows a <hello> message that the agent sends to a client:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>
      urn:ietf:params:netconf:base:1.0
    </capability>
    <capability>
      urn:ietf:params:netconf:capability:candidate:1.0
    </capability>
  </capabilities>
  <session-id>4</session-id>
</hello>
```

These sections about NETCONF are covered:

- [Starting a NETCONF Session, page 15-139](#)
- [Ending a NETCONF Agent Session, page 15-140](#)
- [Starting an SSH NETCONF Session, page 15-140](#)
- [Ending an SSH NETCONF Agent Session, page 15-141](#)
- [Configuring a NETCONF agent, page 15-141](#)
- [Limitations of NETCONF in Cisco IOS XR, page 15-142](#)

Starting a NETCONF Session

To start a NETCONF session, enter the **netconf** command from the exec prompt (through telnet or SSH).

This example shows how to start a TTY NETCONF agent session:

```
client(/users/ore)> telnet 1.66.32.82
Trying 1.66.32.82...
Connected to 1.66.32.82.
```

Escape character is '^]'.
 ^Z

User Access Verification

```
Username:
Password:
RP/0/1/CPU0:Router# netconf echo format
<?xml version="1.0" encoding="UTF-8" ?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>
      urn:ietf:params:netconf:base:1.0
    </capability>
    <capability>
      urn:ietf:params:netconf:capability:candidate:1.0
    </capability>
  </capabilities>
  <session-id>4</session-id>
</hello>]]>]]>
```

When a new session is created, the NETCONF agent immediately sends out a <hello> message with capabilities. At the end of each message transmission, the NETCONF agent sends the EOD marker '<]]>]]>'

The NETCONF agent does not display a prompt like the XML agent does (XML>).

The NETCONF TTY agent does not echo back the received messages and does not format returning messages by default. These capabilities can be added by using the 'echo' and 'format' options.

The client is also required to send a <hello> message with capabilities.

Ending a NETCONF Agent Session

Unlike the XML agent, the client ends the session by sending a <close-session> request.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <close-session/>
</rpc>]]>]]>
```

The agent replies with an <ok> tag and then closes the session.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rpc-reply message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>]]>]]>
```

Starting an SSH NETCONF Session

This example shows how to start an SSH NETCONF agent session:

```
client(/users/ore)> ssh lab@1.66.32.82
lab@1.66.32.82's password:
RP/0/1/CPU0:gsrb#netconf echo format
<?xml version="1.0" encoding="UTF-8" ?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
```

```

    <capability>
      urn:ietf:params:netconf:base:1.0
    </capability>
  </capabilities>
  <session-id>4</session-id>
</hello>]]>]]>

```

The client can also directly start a NETCONF session by specifying the **netconf** command on the ssh command line:

```

client(/users/ore)> ssh lab@1.66.32.82 netconf echo format
lab@1.66.32.82's password:
<?xml version="1.0" encoding="UTF-8" ?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>
      urn:ietf:params:netconf:base:1.0
    </capability>
    <capability>
      urn:ietf:params:netconf:capability:candidate:1.0
    </capability>
  </capabilities>
  <session-id>4</session-id>
</hello>]]>]]>

```

Ending an SSH NETCONF Agent Session

This example shows how to end an SSH NETCONF agent session:

```

<?xml version="1.0" encoding="UTF-8" ?>
<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <close-session/>
</rpc>]]>]]>

```

The agent replies with an **<ok>** tag and then closes the session.

```

<?xml version="1.0" encoding="UTF-8" ?>
<rpc-reply message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>]]>]]>

```

Configuring a NETCONF agent

To configure a NETCONF TTY agent, use the **netconf agent tty** command.

Use the *throttle* and *session timeout* parameters as you would with the XML TTY agent.

```

netconf agent tty
  throttle (memory | process-rate)
  session timeout

```

To enable the NETCONF SSH agent, use this command:

```

ssh server v2
netconf agent tty

```

Limitations of NETCONF in Cisco IOS XR

This sections identifies the limitations of NETCONF in Cisco IOS XR Software.

Configuration Datastores

Cisco IOS XR supports these configuration datastores:

- <running>
- <candidate>

Cisco IOS XR does not support the <startup> configuration datastore.

Configuration Capabilities

Cisco IOS XR supports these configuration capabilities:

- Candidate Configuration Capability
urn:ietf:params:netconf:capability:candidate:1.0

Cisco IOS XR does not support these configuration capabilities:

- Writable-Running Capability
urn:ietf:params:netconf:capability:writable-running:1.0
- Confirmed Commit Capability
urn:ietf:params:netconf:capability:confirmed-commit:1.0

Transport (RFC4741 and RFC4742)

These transport operations are supported:

- Connection-oriented operation
- Authentication
- SSH Transport—Shell based SSH. IANA-assigned TCP port <830> for NETCONF SSH is not supported.
- Other transport

Subtree Filtering (RFC4741)

NETCONF has these subtree filtering limitations in Cisco IOS XR:

- Namespace Selection—Filtering based on specified namespace. This is not supported because Cisco IOS XR does not publish schema name spaces.
- Attribute Match Expressions—Filtering is done by matching a specified attribute value. This filtering with the “Match” attribute can be specified only in Table classes. See this example:

```
<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
```



```

    <running/>
  </source>
</filter>
  <Configuration>
    <InterfaceConfigurationTable>
      <InterfaceConfiguration>
        <Naming>
          <Active>act</Active>
          <InterfaceName Match="GigabitEthernet.*" />
        </Naming>
      </InterfaceConfiguration>
    </InterfaceConfigurationTable>
  </Configuration>
</filter>
</get-config>
</rpc>

```

- **Containment Nodes**—Filtering is done by specifying nodes (classes) that have child nodes (classes). This filtering is by specifying container classes. See this example:

```

<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter>
      <Configuration>
        <InterfaceConfigurationTable/>
      </Configuration>
    </filter>
  </get-config>
</rpc>

```

- **Selection Nodes**—Filtering is done by specifying leaf nodes. This filtering specifies leaf classes. See this example:

```

<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter>
      <Configuration>
        <InterfaceConfigurationTable>
          <InterfaceConfiguration>
            <Naming>
              <Active>act</Active>
              <InterfaceName>GigabitEthernet0/3/0/1</InterfaceName>
            </Naming>
            <Shutdown/>
          </InterfaceConfiguration>
        </InterfaceConfigurationTable>
      </Configuration>
    </filter>
  </get-config>
</rpc>

```

- **Content Match Nodes**—Filtering is done by exactly matching the content of a leaf node. This filtering is done by specifying naming the class value for table classes. See this example:

```

<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>

```

```

</source>
<filter>
  <Configuration>
    <InterfaceConfigurationTable>
      <InterfaceConfiguration>
        <Naming>
          <Active>act</Active>
          <InterfaceName>Loopback0</InterfaceName>
        </Naming>
      </InterfaceConfiguration>
    </InterfaceConfigurationTable>
  </Configuration>
</filter>
</get-config>
</rpc>

```

According to the RFC, a request using an empty content match node should return all <Naming> elements of all entries of the table.

For example, for this request, the response should return <Naming> elements of all the entries of <InterfaceConfigurationTable>:

```

<rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter>
      <Configuration>
        <InterfaceConfigurationTable>
          <InterfaceConfiguration>
            <Naming/>
          </InterfaceConfiguration>
        </InterfaceConfigurationTable>
      </Configuration>
    </filter>
  </get-config>
</rpc>

```

In Cisco IOS XR, this request is not supported and is errored out.

Protocol Operations (RFC4741)

These protocol operations are supported in Cisco IOS XR:

- get—Root level query that returns both the entire configuration and state data is not supported
- get-config
- edit-config
- lock
- unlock
- close-session
- commit (by the Candidate Configuration Capability)
- discard-change (by the Candidate Configuration Capability)

Event Notifications (RFC5277)

Event notifications are not supported in Cisco IOS XR.



CHAPTER 16

Cisco IOS XR Perl Scripting Toolkit

This chapter describes the Cisco IOS XR Perl Scripting Toolkit as an alternative method to existing router 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 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.

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

- [Cisco IOS XR Perl Scripting Toolkit Concepts, page 16-148](#)
- [Security Implications for the Cisco IOS XR Perl Scripting Toolkit, page 16-148](#)
- [Prerequisites for Installing the Cisco IOS XR Perl Scripting Toolkit, page 16-148](#)
- [Installing the Cisco IOS XR Perl Scripting Toolkit, page 16-149](#)
- [Using the Cisco IOS XR Perl XML API in a Perl Script, page 16-150](#)
- [Handling Types of Errors for the Cisco IOS XR Perl XML API, page 16-150](#)
- [Starting a Management Session on a Router, page 16-150](#)
- [Closing a Management Session on a Router, page 16-152](#)
- [Sending an XML Request to the Router, page 16-152](#)
- [Using Response Objects, page 16-153](#)
- [Using the Error Objects, page 16-154](#)
- [Using the Configuration Services Methods, page 16-154](#)
- [Using the Cisco IOS XR Perl Data Object Interface, page 16-157](#)
- [Cisco IOS XR Perl Notification and Alarm API, page 16-166](#)
- [Examples of Using the Cisco IOS XR Perl XML API, page 16-170](#)

Cisco IOS XR Perl Scripting Toolkit Concepts

Table 16-1 describes the toolkit concepts. Some sample scripts are modified and show how to use the API in your own scripts.

Table 16-1 List of Concepts for the IOS XR Perl Scripting Toolkit

Concept	Definition
Cisco IOS XR Perl XML API	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.

Security Implications for the Cisco IOS XR Perl Scripting Toolkit

Similar to using the 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 these specific standard Perl modules to use 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. This 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 this location: <http://www.cpan.org/>

These 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 sample 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 these steps:

Step 1 Extract the contents from the directory in which the file resides by entering this command:

```
tar -f Cisco-IOS_XR-Perl-Scripting-Toolkit-<version>.tar.gz -xzc <destination>
```

Table 16-2 defines the parameters.

Table 16-2 Toolkit Installation Directory Parameters

Parameter	Description
<version>	Defines the version of the toolkit to install, for example, version 1.0.
<destination>	Specifies the existing directory in which to create the toolkit installation directory. A directory called <code>Cisco-IOS_XR-Perl-Scripting-Toolkit-<version></code> 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 this 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 this example:

```
make
```

Step 4 Use the **make install** command, as shown in this 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.

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 this 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 the [“Creating Data Objects” section on page 16-159](#).

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

These 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 the [“Using the Error Objects” section on page 16-154](#).
- 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 16-3](#) lists the optional parameters specified as arguments.

Table 16-3 Argument Definitions

Name	Description
<i>use_command_line</i>	Controls whether or not the <code>new()</code> 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>	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 <code>Term::ReadKey</code> module must be installed. The most secure way of using the toolkit is not to have the input echoed to the screen, which avoids hard coding or any record of passwords being used. The default value is false, which means that the script does not ask for user input. As a command-line option, the <code>interactive</code> argument does not take any arguments. You can specify <code>-interactive</code> to turn on the interactive mode.

Table 16-3 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 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> .

This 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);
```

Alternatively, the arguments can be specified in a file. For example:

The contents of `/usr/trice/perlxml.cfg`:

```
[myrouter]
transport = telnet
host = router1
username = john
password = smith
connection_timeout = 3
```

In the script, the file and profile name are specified:

```
use Cisco : : IOS_XR;
my $session = new Cisco: :IOS_XR(config_file =>
    '/usr/trice/perlxml.cfg',
    profile => 'myrouter');
```

Table 16-4 describes the additional command-line options that can be specified.

Table 16-4 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.

To use the command-line options when invoking a script, use the `-option value` (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. This 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.

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 this 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.

This 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);
```

This 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.

Retrieving the Response XML as a String

This example shows how to use the `to_string` method:

```
$xml_response_string = $response->to_string;
```

Writing the Response XML Directly to a File

This 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

This 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 is represented in an object tree structure. For more information, see this URL:

<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

This example shows how to determine whether an error has occurred while processing a request:

```
my $error = $response->get_error;
if (defined($error)) {
    die $error;
}
```

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 are found.

Retrieving a List of All Errors Found in the Response XML

This example shows how to list all errors that occur, 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 the “Using the Error Objects” section on page 16-154.

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 16-5 lists the methods for the object.

Table 16-5 List of Methods for the Object

Method	Description
<code>get_message</code>	Returns the error message string that was found in the XML.
<code>get_code</code>	Returns the corresponding error code.
<code>get_element</code>	Returns the tag name of the XML element in which the error was found.
<code>get_dom_node</code>	Returns a reference to the element node in the response DOM ¹ tree.
<code>to_string</code>	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. This example displays all information in an error: 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

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.

Committing the Target Configuration

The `config_commit()` function takes these optional arguments:

- *mode*
- *label*
- *comment*
- *Replace*
- *KeepFailedConfig*
- *IgnoreOtherSessions*
- *Confirmed*

This example shows how to use the `config_commit` function:

```
$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 this example:
$commit_id = $response->get_commit_id();
```

Locking and Unlocking the Running Configuration

This example shows how to use the `config_lock` and `config_unlock` functions, which takes no arguments:

```
$error = $session->config_lock;  
$error = $session->config_unlock;
```

Loading a Configuration from a File

This example shows how to contain a filename as an argument:

```
$error = $session->config_load(Filename => 'test_config.cfg');
```

Loading a Failed Configuration

This 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

This 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

This example shows how to use the `config_clear` function, which takes no arguments:

```
$error = $session->config_clear;
```

Getting a List of Recent Configuration Events

This example shows how to use the `config_get_history` function that uses the optional arguments *Maximum*, *EventType*, *Reverse*, and *Detail*:

```
$response = $session->config_get_history(EventType => 'All', Maximum =>10, Detail =>  
'true');
```

It returns a Response object, on which the method *get entries* can be called.

Getting a List of Recent Configuration Commits That Can Be Rolled Back

This example shows how to use the `config_get_commitlist` function that uses the optional arguments *Maximum* and *Detail*:

```
$response = $session->config_get_commitlist (Maximum => 10, Detail => 'true');
```

It returns a Response object, on which the method *get entries* can be called. This returns an array of Entry objects, on which the method *get key* can be called to retrieve the CommitID, and *get data* to retrieve the rest of the fields.

Loading Changes Associated with a Set of Commits

This example shows how to use the `config_load_commit_changes` function to load into the target configuration the changes that were made during one or more commits, and it uses one of three possible arguments: *ForCommitID*, *SinceCommitID*, or *Previous*:

```
$error = $session ->config_load_commit_changes (ForCommitID => 1000000072);
#Loads the changes that were made in commit 1000000072

$error = $session ->config_load_commit_changes (SinceCommitID => 1000000072);
#Loads the changes made in commits 1000000072, 1000000073...up to latest

$error = $session ->config_load_commit_changes (Previous => 4);
#Loads the changes made in the last 4 commits
```

Rolling Back to a Previous Configuration

This example shows how to use the `config_rollback()` function that uses the optional arguments *Label* and *Comment*, and exactly one of the two arguments *CommitID* or *Previous* or *takes only TrialConfiguration*:

```
$error = $session->config_rollback(Label => 'Rollback test', CommitID => 1000000072);
```

Loading Changes Associated with Rolling Back Configuration

This example shows how to use the `config_load_rollback_changes` function to load into the target configuration the changes that would be made if you were to roll back one or more commits. The function uses one of three arguments: *ForCommitID*, *ToCommitID* and *Previous*. For example:

```
$error = $session->config_load_rollback_changes (ForCommitID => 1000000072)
# Loads the changes that would be made to rollback commit 1000000072

$error = $session->config_load_rollback_changes (ToCommitID => 1000000072);
# Loads the changes that would be made to rollback all commits up to and including commit
1000000072
```

Getting a List of Current Configuration Sessions

This example shows how to use the `config_get_sessions` function that uses the optional argument *Detail* to return detailed information about configuration sessions. For example:

```
$response = $session->config_get_sessions (Detail => 'true');
```

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` can be called to retrieve the session ID, and `get_data` method to retrieve the rest of the fields.

Clearing Configuration Session

This example shows how to use `config_clear_session` function that accepts a configuration session ID *SessionID* as argument and clears that configuration session:

```
$error=$session->config_clear_sessions (SessionID => '00000000-000a00c9-00000000');Sending
a Command-Line Interface Configuration Command
```

This 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, use the `get_cli_response()` method, as shown in this example:

```
$response_text = $response->get_cli_response();
```

**Note**

Apart from the `config_commit`, `config_get_history`, `config_get_commitlist`, `config_get_sessions` and `config_cli` methods, each of the other methods return a reference to an error object if an error occurs or is undefined. For more information, see the [“Using the Error Objects”](#) section on page 16-154.

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, knowledge of the underlying management data hierarchy is required. The management data on an Cisco IOS XR router are 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 Cisco IOS XR router. This 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};
```

These sections describe how to use the Perl Data Object Documentation:

- [Understanding the Perl Data Object Documentation, page 16-158](#)
- [Generating the Perl Data Object Documentation, page 16-158](#)
- [Creating Data Objects, page 16-159](#)
- [Specifying the Schema Version to Use When Creating a Data Object, page 16-161](#)
- [Using Data Operation Methods on a Data Object, page 16-161](#)
- [Using the Batching API, page 16-164](#)
- [Displaying Data and Keys Returned by the Data Operation Methods, page 16-165](#)
- [Specifying the Session to Use for the Data Operation Methods, page 16-166](#)

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 below which root object that file's data lies. For example, a file containing configuration data usually ends in `_cfg.html`. Some files may not contain any object definitions, but just some datatypes or filter definitions and 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 these 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 the [“Specifying the Schema Version to Use When Creating a Data Object”](#) section on page 16-161

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 Cisco IOS XR software that you have installed on the router.

To generate the Perl Data Object Documentation:

-
- Step 1** From the perl subdirectory under the extracted contents of the previously 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 are used to generate the HTML documentation.
- Step 2** From the perl subdirectory under the extracted contents of the previously 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.

(The default .html subdirectory already contains two files that were extracted with the toolkit distribution: `root_objects.html` and `common_datatypes.html`. These files are automatically copied to the selected .html directory, if a non-default directory is selected, upon performing this step).

Step 3 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.

Step 4 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 as in this example:

```
<?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 rerun the script.

Linked HTML files are created in the selected (or default) `html` directory. 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 these objects:

- Configuration
- Operational
- Action
- AdminOperational
- AdminAction

This 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. This example shows how to import the Configuration and Operational functions:

```
use Cisco::IOS_XR qw(Configuration Operational);
```

This 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 the use of `Cisco::IOS_XR qw(Configuration Operational)` and refer to the objects simply as Configuration. This may not have the desired effect due to the ambiguity. Instead, you have to refer to them with the more lengthy (that is, fully qualified) syntax, `Cisco::IOS_XR::Data::Configuration`.

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 this 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 is created is known as its parent, as shown in this 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. This 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. This 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');
```

Some table keys have a child object and use brackets { } to indicate the child objects of the key. For example, use this CLI to create an object that corresponds to a router static entry:

```
router static
  address-family ipv4 unicast
    0.0.0.0/0.12.25.0.1
  !
!
```

```
my $router_static = Configuration->RouterStatic->DefaultVRF;
my $static_ipv4 = $router_static->AddressFamily->VRFIPv4->VRFUnicast;
my $static_prefix = $static_ipv4->VRFPrefixTable->VRFPrefix(Prefix => {IPV4Address
=> '0.0.0.0'}, Length => '0');
my $route = $static_prefix ->VRFRouteTable;
my $nexthop = $route->VRFNextHopInfoTable->VRFNextHopInfo(Address => {IPV4Address =>
'12.25.0.1'});
```

Keys can also be specified by passing a hash structure as an argument. The hash structure would usually be returned as a key from one of the data operation methods; for example, *get_keys*, but can be defined explicitly, as in this 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. This 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 the [“Understanding the Perl Data Object Documentation”](#) section on page 16-158.

Specifying the Schema Version to Use When Creating a Data Object

To specify which version of a particular schema you are using, 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.

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 example, you may want to specify that using BGP schema version 1.4. This example shows how to create a BGP object. (The version numbers shown are hypothetical. Substitute the actual version numbers when using this command.)

```
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 Data Operation Methods on a Data Object

To access management data on the router, data operation methods, which can be called on data objects, are provided for the getting, setting, and deletion of 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.

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 the [“Using Response Objects”](#) section on page 16-153. 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.

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`.

This 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 only for 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 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, this 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, to include any nonactive interface configuration in the above example, use this syntax:

```
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.

This example shows how to get the keys for each item in the configuration table, and then for each key, how 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...
    }
}
```

These 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 this 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.

This 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.

This example shows how to add an `IPv4Multicast` object to the `GlobalAFTable` of `BGP AS 1` object:

```
my $data = {'Enabled' => 'true'};
my $bgp_af = Configuration->BGP->AS('AS' => 0)->FourByteAS('AS'=>1);
my $global_af = $bgp_as ->DefaultVRF->Global->GlobalAFTable->GlobalAF ('AF' =>
'IPv4Multicast');
    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.

This example shows how data 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 this example:

```
my $bgp as = Configuration->BGP->AS('AS' => 0)->FourByteAS('AS' => 1);
my $error = $bgp_as->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 the operation was successful or not. 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.

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. This 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 this 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, 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. This is the code:

```
my $response = Configuration->InterfaceConfigurationTable
    ->InterfaceConfiguration(Active => 'act', Name => 'POS0/2/0/0')
    ->get_data;
print $response->get_data;
```

This 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.

This 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.

These sections describe how to use the Alarm API:

- [Registering for Alarms, page 16-166](#)
- [Deregistering an Existing Alarm Registration, page 16-167](#)
- [Deregistering All Registration on a Particular Session, page 16-167](#)
- [Receiving an Alarm on a Management Session, page 16-167](#)

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.

This 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 (cancel this registration). The registration ID can be extracted from the response object by calling the `get_registration_id` method, as shown in this 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. 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 this example:

```
my $alarm = $session->alarm_receive(60); # Wait 60 seconds for an alarm
```

If an alarm is received within the timeout limit, the function returns an alarm object from which these 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 this 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

These sections describe how to control debug and logging facilities within your script:

- [Debug Facility Overview, page 16-168](#)
- [Logging Facility Overview, page 16-169](#)

For more information on how to control debug and logging from the command line when starting a script, see the “[Starting a Management Session on a Router](#)” section on page 16-150.

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 messages to be displayed and which ones not to be displayed.



Note

Debug applies to the script as a whole rather than to each management session.

[Table 16-6](#) lists the current built-in types.

Table 16-6 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 this 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 this 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 this 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 this example:

```
Cisco::IOS_XR::add_debug_types('general' => 'General', 'detailed' => 'Detailed');
```

These types can immediately be used to write debug messages, as shown in this 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 this example:

```
Cisco::IOS_XR::logging_on('transport', 'xml');
```

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 this example:

```
Cisco::IOS_XR::logging_off('xml');
```

By default, the messages are 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, to specify a different log file before carrying out operations on a different management session, see this 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 these arguments when calling `Cisco::IOS_XR::new`, as listed in [Table 16-7](#).

Table 16-7 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

These sections provide examples of using the Cisco IOS XR Perl XML API to perform some of the common router management tasks:

- [Configuration Examples, page 16-170](#)
- [Operational Examples, page 16-177](#)

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 running the script.

You may also need to give yourself execute permission on the script if it is not already set using the `chmod` command:

```
chmod +x <script name>.pl
```

You should be able run the script using this 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 this 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 this 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 this 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');
```

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. This 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 the 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, which is then called, as shown in this 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 this 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

This 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 this example:

```
my $bgp_as = Configuration->BGP->AS(AS => 0)->FourByteAS(AS => 1);
my $bgp_entity = $bgp_as->DefaultVRF->BGPEntity
my $neighbor = $bgp_entity->NeighborTable->
```

```
Neighbor(IPAddress => {IPV4Address => '1.2.3.4'});
```

This example shows how to set the remote autonomous system (AS) number for the neighbor:

```
$error = $neighbor->RemoteAS->set_data(44);
```

To set the description for this neighbor, see this 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 can perform this task using the CLI. You would have to enter a series of commands, as shown in this 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
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 this 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. In this example, CLI commands are 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, you must define the ISIS instance, as shown in this example:

```
my $instance = Configuration->ISIS->InstanceTable->Instance(InstanceID => 1);
```

```
Enable the instance:
$instance->Running->set_data('True');
```

This 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(Enable => 'True');
```

This example shows how to set up the interface:

```
my $if = $instance->InterfaceTable->Interface(Name => 'POS0/2/0/0');
$if->Running->set_data('True');
```

This 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. This example shows how to create an object that corresponds to ISIS configuration:

```
my $isis = Configuration->ISIS;
```

The only child object of ISIS is `InstanceTable`, which has entries of the object instance. Under the `Keys` heading, notice that `Instance` has only one key called `InstanceID`. If the ISIS instance in which you are interested is 1, you can now create a data object corresponding to that instance by specifying the instance ID as an argument. The creation of the data object is shown in this 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 this 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 this 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 this 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;
```

You might not want to perform as many individual steps as are shown in the previous examples, and it is probably not necessary. In practice, you may perform some of the steps at once. This sample code does the same thing in four lines as the above set of examples does in seven lines:

```
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. These 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, this 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 1.1.1.1
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 and ensure that it is started. This is shown in this example:

```
my $ospf_process = Configuration->OSPF->ProcessTable->Process(InstanceName => '1');
$ospf_process->Start->set_data('true');
```


This example shows how to set up the router ID for the process:

```
$ospf_process->DefaultVRF->RouterID->InterfaceID->set_data('1.1.1.1');
```

This example shows how to set up the area of which this interface is part:

```
my $area = $ospf_process->DefaultVRF->AreaTable->Area(IntegerID => 1);
$area->Running->set_data('true');
```

This 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**. This example shows where you would use the `get_keys` function:

```
my $response = Configuration->AAA->UsernameTable->get_keys;
my @keys = $response->get_keys;
```

You could use the resulting array however you want; for example, to display your own compact list of usernames. This is shown in this 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 this 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

These 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
```

These 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. This example shows how to define the relevant tables that you are interested in:

```
my $acl_table = Configuration->IPV4_ACLAndPrefixList;
my $if_conf_table = Configuration->InterfaceConfigurationTable;
```

This 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->AccessListTable->AccessList(Name => 'user1');
$error = $acl->AccessListEntryTable->
    AccessListEntry(SequenceNumber => '50')->ACERule ->
    set_data(SourceAddress => '1.2.3.4',
            Grant => 'Deny',
            Protocol => 'IP');
```

This 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(Name => '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 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 this list:

- 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) that you enter when prompted are included. 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.
- 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.

Configuring a New Static Route Entry

This example shows how to add a new static route entry with the applicable CLI commands:

```
router static
  address-family ipv4 unicast
    0.0.0.0/0 12.25.0.1
```

This script is used for the data object interface:

```
my $static_route = Configuration->RouterStatic->DefaultVRF;
my $static_ipv4 = $static->AddressFamily->VRFIPv4->VRFUnicast;
my $static_prefix = $static_ipv4->VRFPrefixTable->
  VRFPrefix(Prefix => {IPV4Address => '0.0.0.0'}, Length => '0');

my $route = $static_prefix->VRFRouteTable;
my $nexthop = $route->VRFNextHopInfoTable->
  VRFNextHopInfo(Address => {IPV4Address => '12.25.0.1'});
$error = $nexthop->Description->set_data('sample');
```

Operational Examples

Certain examples can be used to retrieve operational data from the router. These examples all make use of the Perl Data Object API because of 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 of these 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.

There are some scripts that give examples of how to display the retrieved data in different ways. There 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 ease of viewing in a web browser. These examples take full advantage of having the response data in XML format. HTML can be produced with ease from XML using the style sheet transformation language XLST.

Retrieving the Operational Information for All Interfaces on the Router

This example shows how to retrieve a single table of data, which can be done with ease by using the Data Object interface `get_entries` function. It can be done all in one line (rather than 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 these steps:

Step 1 Define the key for the data node in which you are interested (for example, primary RP), as shown in this example:

```
my $data_node = {RPLocation => {Rack => 0, Slot => RP0, Instance => 'CPU0'}};
```

Step 2 Define the table and what contents to retrieve, as shown in this 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 this example:

```
my $result = $interface_table->get_entries;
```

Step 4 Check to see if an error occurred. If not, retrieve the entries, as shown in this 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. The last two stages are exactly the same.

To retrieve the link state database for a particular level for ISIS, perform these steps:

Step 1 Define the table from which you want to get data from, as shown in this 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 this example:

```
my $result = $table->get_entries;
```

Step 3 Check to see if an error occurred. If not, retrieve the entries, as shown in this 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 other associated data with them; this cannot be done using the CLI **show** commands.

To get a list of all interfaces on the system, perform these steps:

Step 1 Define the table, as shown in this 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 this example:

```
my $result = $interface_table->get_keys;
if ($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.

This 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 **show interfaces** CLI command.

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.

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. The hostname is displayed by the **show isis neighbors** CLI show command, 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, this piece of 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 the [“Retrieving the Combined Interface and IP Information for Each Interface” section on page 16-179](#). 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 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 this example:

```
my $response =
    Operational->VRFTTable->VRF(VrfName='VRF1')->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.

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 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, display the operational data for each interface on the router:

Step 1 Use the `require` statement to specify the name of the module, as shown in this example:

```
require "xml_to_html_table.pm"; # may need to specify a path here
```

Step 2 Retrieve the information in XML format. This 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 => RP0, Instance => "CPU0"})->SystemView->InterfaceTable;
my $response_string = $interface_table->find_data->to_string;
```

Step 3 Transform the XML to HTML, as shown in this 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 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 the [“Displaying the Interface State in a Customized HTML Table”](#) section on page 16-182.

Displaying the Interface State in a Customized HTML Table

In many cases, the generic HTML table example does not display the information exactly the way you would 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. These 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; this makes it easy to spot problems without having to scroll down a long list.

The transformation is again done using XLST. 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 **show bgp neighbors** command, 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; this includes the address family information that is displayed in a simple subtable.

The benefits of having the information in HTML are:

- Format of bold headings makes the information clearer.
- 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.

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.

Performing Actions Whenever Certain Events Occur

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, someone is informed through e-mail (network administrator) about the events listed in [Table 16-8](#).

Table 16-8 List of Events

Event	Description
Interfaces going up/down	When the event occurs, an e-mail is sent informing the recipient of the interface, the router on which the interface is, 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 <code>commitID</code> 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.

These 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 can be a regular e-mail or a message sent to a pager. This is not practical for a long message (for example, a configuration change), but can be well-suited to a single-line message similar to the interface up or down case.

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.



CHAPTER 17

Sample BGP Configuration

This excerpt displays the relevant portions of the 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
    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
  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
```




GLOSSARY

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.

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.

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

- 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*.
- 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.
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 TTY-based XML agent 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.



INDEX

Symbols

- <Action> [4-54](#)
- <AdminAction> [4-54](#)
- <AdminOperational> [4-54](#)
- <Alarm> [10-117](#)
- <Clear> [1-9](#)
- <ClearConfigurationInconsistency> [1-9](#)
- <ClearConfigurationSession> [1-9](#)
- <CLI> [1-9](#)
- <CLI> tag [2-45](#), [3-49](#), [6-91](#)
- <Client> [2-41](#)
- <ClientName> [2-43](#)
- <Comment> [2-41](#)
- <Commit> [1-9](#), [2-22](#)
 - Comment attribute [2-23](#)
 - Confirmed attribute [2-23](#)
 - errors [2-25](#)
 - IgnoreOtherSessions attribute [2-24](#)
 - KeepFailedConfig attribute [2-23](#)
 - Label attribute [2-23](#)
 - Mode attribute [2-23](#)
 - Replace attribute [2-24](#)
 - Rollback [2-33](#)
- <Commit> operation [2-27](#)
- <CommitId> [2-41](#)
- <CommitId> tag [2-25](#), [2-33](#)
- <Configuration> [4-54](#)
- <Configuration/ > tag [5-68](#)
- <Delete> [1-8](#), [2-25](#), [4-53](#)
 - AAA privileges [8-102](#)
 - native data operations [4-59](#)
- <Delete/ > tag [4-65](#)
- <destination> [16-149](#)
- <Detail> [2-41](#)
- <EBGPMultihopMaxHopCount> [5-72](#)
- <Elapsed> [2-43](#)
- <Error> element [8-102](#)
- <FailedConfig> tag [2-27](#)
- <File> [2-19](#)
- <Filter> [5-85](#)
- <Get> [1-8](#), [2-15](#), [2-17](#), [3-49](#), [4-53](#)
 - AAA privileges [8-102](#)
 - native data operations [4-59](#)
 - triggering [4-58](#)
- <GetConfigurationCommitList> [1-9](#)
- <GetConfigurationHistory> [1-9](#), [2-38](#)
 - maximum attribute [2-38](#)
- <GetConfigurationSession> [1-9](#)
- <GetConfigurationSessions> [2-43](#)
- <GetDataSpaceInfo/ > tag [4-66](#)
- <GetDataSpaceInformation> [1-8](#)
- <GetNext> [1-9](#)
 - IteratorID [7-94](#)
- <GetVersionInfo/ > tag [4-66](#)
- <GetVersionInformation> [1-8](#)
- <HoldTime> [4-58](#)
- <Label> [2-41](#)
- <Line> [2-41](#), [2-43](#)
- <Load> [1-9](#), [2-19](#), [2-27](#), [3-52](#)
- <Lock> [1-9](#), [2-15](#)
- <LockHeld> [2-43](#)
- <LoopbackCheck> [5-72](#)
- <Maximum> [2-41](#)
- <Naming> tag [4-58](#)
- <Node> [2-43](#)

<Operational> 3-49, 4-54
 <Previous> 2-34
 <Process> 2-43
 <ProcessID> 2-43
 <Register> 10-117
 <RemoteAS> 5-72
 <Response>
 IteratorID 7-94
 <Rollback> 1-9, 2-25, 2-33, 2-34
 <Save> 1-9, 2-19, 2-21
 <SessionId> 2-43
 <Set> 1-8, 2-25, 2-45, 4-53
 AAA privileges 8-102
 native data operations 4-59
 <Since> 2-43
 <Timestamp> 2-41
 <Unlock> 1-9, 2-28
 <Unlock/ > 2-28
 <User> 2-41
 <UserId> 2-43
 <version> 16-149

A

AAA (authentication, authorization, and accounting)
 authorization 8-101
 definition 1-2
 security (perl scripting toolkit) 16-148
 access control list
 See ACL 8-105
 ACL 8-105
 ACL (Access Control List)
 CLI commands 16-176
 entry, add 16-176
 inbound traffic 16-176
 list 16-176
 perl data object API 16-176
 add_neighbors_to_group.pl file 16-173
 alarm_deregister function 16-167

alarm_operations.xsd 14-136
 alarm_receive function 16-167
 alarm_register function 16-166
 alarms
 deregistration 10-118
 filter criteria, types of 10-117
 notification 10-119
 registration 10-117
 tags, types of 10-119
 API (application programming interface)
 perl data object 16-148
 perl notification/alarm 16-148
 perl XML
 concept 16-148
 configuration examples 16-171
 operational examples 16-177
 arguments, management session
 connection_timeout 16-151
 host 16-151
 interactive 16-150
 password 16-151
 port 16-151
 prompt 16-151
 response_timeout 16-151
 ssh_version 16-151
 transport 16-151
 use_command_line 16-150
 username 16-151
 Atomic mode 2-23

B

BASE package common schemas 14-136
 batch_send method 16-165
 batch_start method 16-164
 batch API
 batch_send method 16-165
 batch_start method 16-164
 usage 16-164

batched requests [1-10](#)
 BestEffort [2-23](#)
 BGP (Border Gateway Protocol)
 CLI commands [16-172](#)
 configuration [17-185](#)
 data object interface [16-172](#)
 get request [3-49](#)
 neighbor
 add list [16-172](#)
 members, display [16-173](#)
 set description [16-172](#)
 bgp_neighbor_table_html.pl file [16-183](#)
 Border Gateway Protocol
 See BGP [3-49](#)
 browse, target configuration [2-15](#)

C

cernno [11-126](#)
 ChangedConfig [2-16](#)
 chmod command [16-170](#)
 CircuitType object [16-174](#)
 Cisco-IOS_XR-Perl-Scripting-Toolkit-.tar.gz file [16-149](#)
 ClearConfigurationInconsistency tag [1-9](#)
 ClearConfigurationSession tag [1-9](#)
 Clear tag [1-9](#)
 CLI (command-line interface)
 defined [1-2](#)
 operations [1-9](#)
 cli_operations.xsd [14-136](#)
 CLI command
 encapsulated [1-3, 1-9](#)
 show [4-54](#)
 show aaa userdb [16-175](#)
 show bgp neighbors [16-182](#)
 show interfaces [16-179, 16-180](#)
 show isis database level [16-178](#)
 show isis neighbors [16-180](#)
 xml agent tty [13-129](#)
 ClientID attribute [1-5](#)
 client session
 commit operation [2-25](#)
 limitation [2-13](#)
 CLI tag [1-9](#)
 Comment [2-23](#)
 comment [2-33](#)
 commit [2-29](#)
 changes [2-29](#)
 database [2-25](#)
 identifier [2-33](#)
 CommitChanges [2-16](#)
 Commit tag [1-9](#)
 common_datatypes.xsd [14-136](#)
 common datatype definitions [14-136](#)
 component-specific schemas [14-135, 14-136](#)
 Comprehensive Perl Archive Network
 See CPAN [16-149](#)
 config_clear_sessions function example [16-156](#)
 config_clear function example [16-155](#)
 config_cli() function example [16-156](#)
 config_commit () function [16-154](#)
 config_commit function example [16-154](#)
 config_get_commitlist function example [16-155](#)
 config_get_history function example [16-155](#)
 config_get_sessions function example [16-156](#)
 config_load_commit_changes function example [16-156](#)
 config_load_failed function example [16-155](#)
 config_load_rollback_changes function example [16-156](#)
 config_load function example [16-155](#)
 config_lock function example [16-155](#)
 config_rollback() function example [16-156](#)
 config_save() function example [16-155](#)
 config_services_operations.xsd [14-136](#)
 config_unlock function example [16-155](#)
 Configuration change event [16-183](#)
 configuration change notification [3-52](#)
 Configuration function [16-174](#)
 configuration history [2-14](#)

Configuration Manager [1-3, 1-9, 8-102](#)
 and error reporting [11-125](#)
 Configuration services [1-3, 1-9, 8-102](#)
 configuration session information [2-14](#)
 Confirmed [2-23](#)
 connection_timeout argument [16-151](#)
 container [5-67, 5-69](#)
 Content attribute [5-67, 5-82](#)
 Count argument [16-163](#)
 Count attribute [5-67, 5-83](#)
 CurrentConfig [2-16](#)
 custom filters [5-67](#)

D

data, display how to
 example [16-165](#)
 get_data function [16-165](#)
 data objects
 create [16-159](#)
 operation methods [16-161](#)
 schema version [16-161](#)
 data operation methods, management session [16-166](#)
 debug facility
 definition, types of [16-168](#)
 disable [16-169](#)
 enable [16-169](#)
 insert message [16-169](#)
 overview [16-168](#)
 debug option [16-152](#)
 declaration
 attributes [1-5](#)
 tag [1-4, 1-5](#)
 delete_data method
 definition [16-164](#)
 example [16-164](#)
 Delete tag [1-8](#)
 deny_access.pl file [16-177](#)
 dependencies [4-58](#)

deregistering, alarms [10-118](#)
 Details [2-38](#)
 display_neighbor_group_members.pl file [16-173](#)
 documentation, perl data object
 definition items [16-158](#)
 overview [16-158](#)
 Document Type Definition [14-135](#)
 DOM (Data Object Model)
 example [16-153](#)
 tree type [16-153](#)
 DTD (Document Type Definition)
 See document type definition [14-135](#)

E

element, null value [4-58](#)
 enable the dedicated agent [13-131](#)
 encoding (UTF-8), XML [1-5](#)
 error attributes [11-122, 11-123](#)
 ErrorCode [11-122](#)
 ErrorMessage [11-122](#)
 error object, methods
 get_code [16-154](#)
 get_dom_node [16-154](#)
 get_element [16-154](#)
 get_message [16-154](#)
 to_string [16-154](#)
 error reporting
 nonexistent data [4-63](#)
 types of [11-121](#)
 event notification [3-51](#)
 Event Type [2-38](#)
 EventType [2-38](#)

F

files, perl scripting toolkit
 add_neighbors_to_group.pl [16-173](#)

bgp_neighbor_table_html.pl [16-183](#)
 Cisco-IOS_XR-Perl-Scripting-Toolkit-.tar.gz [16-149](#)
 deny_access.pl [16-177](#)
 display_neighbor_group_members.pl [16-173](#)
 generic_interface_props_table.pl [16-181](#)
 get_ip_interfaces.pm [16-179](#)
 interface_props_table.pl [16-182](#)
 interface_props_table.xsl [16-182](#)
 ios_xr_log.txt [16-169](#)
 list_isis_neighbors.pm [16-180](#)
 notification.pl [16-184](#)
 show_bgp_neighbors.pl [16-181](#)
 show_ip_interfaces.pl [16-180](#)
 xml_to_html_table.pm [16-181](#)
 xml_to_html_table.xsl [16-181](#)
 filter, criteria types [10-117](#)
 Filter argument [16-163](#)
 find_data function [16-175](#)
 find_data method
 definition [16-162](#)
 example [16-162](#)

G

generic_interface_props_table.pl file [16-181](#)
 get_code method [16-154](#)
 get_commit_id() method example [16-155](#)
 get_data method
 definition [16-161](#)
 example [16-161](#)
 get_dom_node method [16-154](#)
 get_dom_tree method [16-176](#)
 get_element method [16-154](#)
 get_entries function [16-178, 16-180](#)
 get_entries method
 definition [16-163](#)
 example [16-163](#)
 get_error method example [16-153](#)
 get_errors method example [16-153](#)

get_ip_interfaces() function [16-179, 16-180](#)
 get_ip_interfaces.pm file [16-179](#)
 get_keys function [16-180](#)
 get_keys method
 definition [16-162](#)
 example [16-162](#)
 get_message method [16-154](#)
 GetConfigurationCommitList tag [1-9](#)
 GetConfigurationHistory tag [1-9](#)
 GetConfigurationSessions tag [1-9](#)
 GetDataSpaceInfo tag [1-8](#)
 GetNext tag operation [1-8, 1-9](#)
 Get tag [1-8](#)
 GetVersionInfo tag [1-8](#)

H

hash structure
 definition [16-157](#)
 example [16-157](#)
 hierarchy
 leaf nodes [4-57](#)
 structure [4-55](#)
 tables [4-55](#)
 host argument [16-151](#)
 HostnameTable object [16-180](#)
 HTML table
 customize, interface state display [16-182](#)
 enhancement list [16-182](#)

I

IgnoreOtherSessions [2-24](#)
 installation, perl scripting toolkit
 directory parameters [16-149](#)
 procedure [16-149](#)
 interactive argument [16-150](#)
 interface_props_table.pl file [16-182](#)

interface_props_table.xml file [16-182](#)

interfaces, get list

- examples [16-179](#)
- procedure [16-179](#)

Interfaces going up/down event [16-183](#)

InterfaceTable object [16-174](#)

ios_xr_log.txt file [16-169](#)

IP address, find interfaces [16-175](#)

IPv4 address family example [16-173](#)

ISIS (Intermediate System-to-Intermediate System)

- circuit type, find [16-173](#)
- CLI commands [16-173](#)
- hostname and interface, list [16-180](#)
- instance ID [16-173](#)
- set up [16-173](#)

ItemNotFound [1-6](#)

ItemNotFoundBelow [1-6](#)

IteratorID [7-94](#)

K

KeepFailedConfig [2-23](#)

keys, display how to

- example [16-165](#)
- get_keys function [16-165](#)

L

Label [2-23](#)

label [2-33](#)

leaf nodes [4-57](#)

leaf object [5-71](#)

link state database, retrieval

- examples [16-178](#)
- procedure [16-178](#)

list_isis_neighbors.pm file [16-180](#)

Load tag [1-9](#)

lock [2-13, 2-14](#)

Lock tag [1-9](#)

log_file option [16-152](#)

logging facility

- arguments, types of [16-170](#)
- disable [16-169](#)
- enable [16-169](#)
- overview [16-169](#)

logging option [16-152](#)

M

make command [16-149](#)

make install command [16-149](#)

Management Plane Protection

- See* MPP [8-104](#)

management session

- close

 - close()method [16-152](#)
 - script [16-152](#)

- data operation methods [16-166](#)
- start

 - arguments [16-150](#)
 - create, object type [16-150](#)

Match attribute [5-75](#)

Maximum [2-38](#)

MergedConfig [2-16](#)

Mode [2-23](#)

modules, perl scripting toolkit [16-148](#)

mpls-te task name [8-103](#)

MPP [8-104](#)

- inband traffic [8-104](#)
- out-of-band traffic [8-104](#)

N

namespace [4-54](#)

native_data_common.xsd [14-136](#)

native_data_operations.xsd [14-136](#)

native data

- access techniques [5-67](#)
- model, types of [1-3](#)
- operations [4-53](#)
- request, nonexistent data [4-63](#)
- tags [1-8](#)

native management data model [1-8](#)

NET (Network Entity Title) example [16-173](#)

nonexistent data [4-63](#)

NotFound [1-6](#)

notification.pl file [16-184](#)

notifications

- alarms [10-119](#)
- list of events [16-183](#)
- steps for script [16-183](#)

null value [4-58](#)

O

object class, hierarchy

- combine [5-67, 5-72](#)
- compressed [5-74](#)
- content [4-53](#)
- duplicated [5-72](#)
- nonexistent data [4-63](#)
- operational [5-71](#)

operation information, retrieval

- examples [16-178](#)
- procedure [16-178](#)

operation processing errors [11-121, 11-125](#)

OperationType attribute [2-17](#)

operation type tag

- CLI [1-9](#)
- configuration services [1-9](#)
- definition [1-8](#)
- native data [1-8](#)
- structure, top-level [1-4](#)

options, command-line

- debug [16-152](#)

- log_file [16-152](#)
- logging [16-152](#)
- telnet_dump_log [16-152](#)
- telnet_input_log [16-152](#)

OSPF (Open Shortest Path First)

- CLI commands [16-175](#)
- configuration [16-175](#)
- router ID [16-175](#)

ouni task name [8-103](#)

P

password argument [16-151](#)

perl scripting toolkit, concepts

- perl data object API [16-148](#)
- perl notification/alarm API [16-148](#)
- perl XML API [16-148](#)

port argument [16-151](#)

privileges, security [8-101](#)

prompt argument [16-151](#)

R

read privileges [8-102](#)

registering, alarms [10-117](#)

repeat naming information [5-67, 5-79](#)

Replace [2-24](#)

request

- <Get>
 - ChangedConfig [2-17](#)
- batching [1-10](#)
- definition [1-2](#)
- maximum size [1-6](#)
- minor and major version numbers [1-5](#)
- repeated naming information [5-79](#)
- tag [1-4](#)
- top level structure of [1-4](#)

Request Type tag [4-54](#)

response

- block size [7-93](#)
- definition [1-2](#)
- error reporting [11-121](#)
- large data retrieval (using iterators) [7-93](#)
- major and minor version numbers [1-5](#)
- minimum [1-6](#)
- namespace declaration in [4-58](#)
- nonexistent data [4-63](#)
- tag [2-24](#)

response_timeout argument [16-151](#)

Reverse [2-38](#)

rollback [2-14](#)

RollbackChanges [2-16](#)

Rollback tag [1-9](#)

router administration, operational data [4-54](#)

RP [8-105, 13-129](#)

running configuration

- browse [2-15](#)
- browsing [2-14](#)
- locking [2-14](#)
- replacing [2-14, 2-45](#)
- target configuration commit [2-22](#)
- unlocking [2-14, 2-28](#)

S

Save tag [1-9](#)

schema file organization [14-136](#)

schemas, XML [14-135](#)

set_data method

- definition [16-163](#)
- example [16-163](#)

Set tag [1-8](#)

show_bgp_neighbors.pl file [16-181](#)

show_ip_interfaces.pl file [16-180](#)

show aaa userdb CLI command [16-175](#)

show bgp neighbors CLI command [16-182](#)

show interfaces CLI command [16-179, 16-180](#)

show isis database level CLI command [16-178](#)

show isis neighbors CLI command [16-180](#)

Source attribute [2-15](#)

SSH

definition [1-2](#)

option [13-129](#)

ssh_version argument [16-151](#)

streaming [7-99](#)

system logging message (syslog) [3-51](#)

T

tag

configuration services operation, types of [1-9](#)

XML [1-3](#)

XML <Response> [1-5](#)

XML API [1-1](#)

XML mapping, types of [12-127](#)

target configuration

browsing [2-14](#)

commit [2-13, 2-14, 2-45](#)

syslog [3-51](#)

commit record [2-25](#)

loading [2-14](#)

modified, uncommitted [2-17](#)

saving to file [2-14, 2-22](#)

TaskGrouping attribute [8-103](#)

task names

mpls-te [8-103](#)

ouni [8-103](#)

telnet_dump_log

argument [16-170](#)

option [16-152](#)

telnet_input_log

argument [16-170](#)

option [16-152](#)

Telnet option [13-129](#)

throttle

cpu [7-99](#)

- memory [7-99](#)
- timestamp [2-29](#)
- to_string method
 - description [16-154](#)
 - example [16-153](#)
 - XML response [16-176](#)
- transport argument [16-151](#)
- transport debug type [16-168](#)
- transport errors [11-121, 11-122](#)
- triggering a <Get> operation [4-58](#)
- TTY transport
 - enable agent, how to [13-129](#)
 - enable session, how to [13-129](#)
 - error code [13-130](#)
 - exit, how to [13-130](#)
 - options
 - SSH [13-129](#)
 - Telnet [13-129](#)

U

- Unlock tag [1-9](#)
- upgrades, schema file [14-137](#)
- use_command_line argument [16-150](#)
- useid [2-29](#)
- user debug type [16-168](#)
- username argument [16-151](#)
- usernames, get list [16-175](#)

V

- version
 - major and minor [9-107](#)
 - mismatch [9-111](#)
 - placement in xml [9-109](#)
 - retrieving [9-113](#)
 - retrieving schema [9-115](#)
 - run-time usage [9-108](#)

- VersionMismatchExists [9-110](#)
- version, XML [1-5](#)
- VersionMismatchExists [9-111](#)
- VersionMismatchExistsBelow [9-111](#)
- virtual route forwarding
 - See* VRF [8-105](#)
- VRF [8-105](#)

W

- wildcards [5-67](#)
- World Wide Web Consortium (W3C) XML Schema Language [14-135](#)
- write_file method example [16-153](#)
- write privileges [8-102](#)

X

- XLST
 - procedure [16-181](#)
 - tabular XML data, display [16-181](#)
- XML (extensible markup language)
 - agent [1-2, 1-3](#)
 - client [1-2](#)
 - instance [4-58](#)
 - operation [1-2](#)
 - operation provider [1-2](#)
 - parse errors [11-121, 11-122](#)
 - schema [1-2](#)
 - definitions for the native data operation type tags [1-8](#)
 - errors [11-121, 11-123](#)
 - session [1-5](#)
 - xml_api_common.xsd [14-136](#)
 - xml_api_protocol.xsd [14-136](#)
 - xml_response_parts debug type [16-168](#)
 - xml_to_html_table.pm file [16-181](#)
 - xml_to_html_table.xsl file [16-181](#)
 - xml agent tty CLI command [13-129](#)

xml debug type [16-168](#)
XML mapping tags [12-127](#)
XML request
 receiving [13-130](#)
 sending [13-130](#)
XML schemas [14-135](#)