JUNOScript: An XML-based Network Management API
draft-shafer-js-xml-api-00

Status of this Memo

This document is an Internet-Draft and is NOT offered in accordance with Section 10 of RFC2026, and the author does not provide the IETF with any rights other than to publish as an Internet-Draft.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on February 25, 2003.

Copyright Notice

Copyright (C) The Internet Society (2002). All Rights Reserved.

Abstract

JUNOScript is an XML-based API for managing devices. It allows access to both operational and configuration data using a simple RPC mechanism. Sessions can be established using a variety of connection-oriented access methods. This document describes the framing protocol, message content, and capabilities of this API. Design decisions and motivations are also discussed. No attempt is made to formally define a protocol, but rather to document the capabilities of JUNOScript as part of the discussion of XML-based network management.
# Table of Contents

1. Introduction .............................................. 4
1.1 Why use XML? ........................................ 4
2. JUNOScript Sessions .................................... 6
2.1 Connection Oriented .................................. 6
2.2 Transport Independent ................................. 6
2.2.1 Access Methods .................................. 7
2.3 JUNOScript Protocol ................................ 9
2.3.1 Session Protocol ................................ 10
2.4 Error Handling ....................................... 11
2.5 Framing Protocol Errors .............................. 12
3. RPC Content ........................................... 13
4. Configuration Operations ............................... 14
4.1 Configuration Data .................................. 14
4.1.1 Configuration in Text Format: `<configuration-text>` 14
4.1.2 Configuration in XML Format: `<configuration>` 15
4.2 Retrieving Configuration with `<get-configuration>` 19
4.2.1 Implementation Note ............................... 21
4.3 Pushing Configuration Data with `<load-configuration>` 21
4.3.1 Disposition of Incoming Configuration .......... 22
4.4 Removing Configuration Data .......................... 25
4.5 Deactivating Configuration Data .................... 26
4.6 Reverting to Prior Configuration Data with Rollback 27
4.7 Database Locking .................................... 28
4.7.1 Acquiring the Exclusive Database Lock ........ 28
4.7.2 Releasing the Exclusive Database Lock .......... 29
4.8 Primitive Operations ................................ 30
4.9 Loading Configuration Patches ....................... 30
4.10 Committing Configuration Data ...................... 31
5. Operational Data ...................................... 33
5.1 Operational RPC Encoding ............................ 33
6. XML Topics ........................................... 35
6.1 Schema Validation ................................... 35
6.2 Per-release schema definitions ..................... 36
6.3 Undocumented Configuration Data .................... 37
6.4 Comments in Configuration Data ..................... 37
6.5 Configuration Groups ................................ 38
6.5.1 Operator Override ................................ 39
6.6 Whitespace Issues .................................. 39
6.7 Namespace Issues ................................... 40
6.8 Encoding Issues .................................... 40
6.8.1 Boolean Attributes ............................... 41
6.9 Push-based Parsers .................................. 42
7. Conclusion ............................................. 43
8. Security .............................................. 45
9. Legal Notices .......................................... 46
References .............................................. 47
1. Introduction

In January 2001, Juniper Networks introduced an Application Programming Interface (API) for the JUNOS network operating system. This API is part of our XML-based Network Management (XNM) effort and is marketed under the trademark JUNOScript. This document describes the protocol used by the API and provides some insight into its design and implementation.

JUNOScript allows full access to both operational and configuration data using a light-weight remote procedure call (RPC) encoded in XML [1].

JUNOScript uses a simple model, designed to minimize both the implementation costs and the impact on the managed device. The model does not require on-box tools such as XSLT [2], since these may limit the size and number of machines that can implement the model. We aimed for simplicity and ease of implementation in most design issues, but not at the expense of expressiveness.

1.1 Why use XML?

In selecting a technology, the central consideration is how well it solves the problems at hand. XML is useful in an API for several reasons:

- Simplicity -- XML can be explained in two paragraphs.
- Hierarchy -- XML’s natural mechanism for expressing hierarchical content is a perfect match for the complex data used in networking.
- Extensibility -- XML content is extensible at both the macro level (adding new operations) and the micro level (adding new elements).
- Supporting technologies -- XML is the core of a rich set of supporting standards, from XSLT to XML:DB [8].
- Tools -- XML has a large catalog of available tools, from Xalan (XSLT) [17] to Batik (PDF generator) [18] to Xindice (XML:DB) [19].
- "Live" data -- Content can be read with, manipulated via, and saved using XML APIs.
- Database reachability -- Networking data comes from, goes to, or ends up in databases. Standard APIs such as XML:DB and proprietary APIs from database vendors such as Oracle and IBM allow data to be retrieved and stored in XML format.

- Standard APIs -- Simple API for XML (SAX) [20] and Document Object Model (DOM) [5] are widely portable.

- Web tie-ins -- Data that is not stored in a database is usually presented directly to users. Passing an Extensible Stylesheet Language stylesheet to web browsers along with XML data enhances presentation.

- Feature velocity -- XML technologies are evolving at a furious pace. New specifications allow interoperability between an increasing set of problem domains.

- Human readable -- The text-based nature of XML makes it easier for users to debug, log, trace, and tune. For example, it is easy to detect by eye that an end tag is missing in a section of XML.

- Stream-oriented -- Large DOM trees will always be troublesome, but recent trends toward streaming APIs makes large XML datasets manageable.

- Filterable -- A stream of SAX events can be filtered or doctored as the stream is generated.

- Wide interest and development effort -- Numerous open source tools use XML for anything from configuration to output. In addition, commercial tools use XML for anything from database access to web site content management.

- Support for all common development languages -- C, Perl, Java, Tcl.

By using XML-based tools, configuration data for an entire network can be retrieved from a database and transformed into a format acceptable by each particular managed device. It can then be handed to the device, the results tallied with those of other devices, and results displayed in multiple presentation formats, such as HTML or PDF.
2. JUNOScript Sessions

JUNOScript uses a simple protocol to issue RPCs to the managed device. This section examines the protocol, its goals and its design.

2.1 Connection Oriented

JUNOScript can be run over any connection-oriented transport mechanism. The connections are long-lived and stateful.

Being connection-oriented allows the use of locks and session status without complex session management. Writing and debugging in environments where session management relies on timeouts leads to complex failure scenarios. Application failure forces either manual cleanup or a delay waiting for session timeouts to expire.

By using a connection, we can know that the connection will be closed in a timely fashion when the client aborts, core dumps, or ends prematurely. The client will close any open sockets when the it exits, including the connection to the server. At that point, the server can perform whatever cleanup is required.

If connectivity is lost between the client and server, TCP keepalives will cause the connection to close after a timeout based on the round-trip time to the client. If connectivity is restored before the timeout expires, recovery is automatic. If not, the client will receive a TCP reset when the next packet (data or keepalive) is sent.

2.2 Transport Independent

JUNOScript runs independent of the transport protocol used. All that is required is a bidirectional stream of byte data. The typical usage scenario is a network connection over in-band or out-of-band networks. Use of out-of-band networks is advised, since failure of the forwarding path will not affect the out-of-band network.

Usage scenarios also include console access via direct serial line and via terminal servers running telnet. Both scenarios are important for users who rely on applications and need to avoid the lower level details of the managed device. These users should stay within the safety of the application even during times of network failure and catastrophe recovery. Forcing users into the CLI at such times negates much of the win brought by network management applications.
### 2.2.1 Access Methods

The supported transport protocols include:

- ssh (Secure Shell)
- ssl (Secure Sockets Layer) and tls (Transport Layer Security)
- clear-text
- serial (console)
- telnet (terminal server attached to console)

#### 2.2.1.1 SSH

SSH [21] is a protocol that invokes a remote shell over a secure connection. Public/private key technology (DSA and RSA) is used to provide an authenticated, encrypted connection between client and server. Use of the ‘ssh-agent’ key-store allows multiple connections to be opened without repeatedly prompting for passphrases.

The JUNOS CLI contains a command ("xml-mode") that converts the login session into a JUNOScript session. After issuing the "xml-mode" command, all subsequent communication is in XML. The ssh client accept this command as part of the command line:

```
ssh my-router xml-mode
```

Using ssh was helpful for both development, deployment, and simplicity. For testing, JUNOScript can easily be accessed by manually invoking ssh to a router or by invoking the "xml-mode" command from an existing CLI session.

Since ssh is already supported and widely deployed, no additional configuration is required to allow JUNOScript-based applications to access deployed devices. Although tunneling an application protocol through ssh may seem awkward, its convenience and utility make it useful.

This access method also allows JUNOScript to be used in a traditional UNIX pipe:

```
make-query --format=XML | ssh my-router xml-mode | xsltproc ...
```
2.2.1.2 SSL

SSL [10] is a protocol for building authenticated, encrypted connections. It uses X.509 [11] certificates to authenticate the client to the system and the system to the client. SSL provides session encryption.

SSL is supported by using the ‘stunnel’ [22] wrapper around the JUNOScript protocol, which includes a simple challenge/response login RPC. The login RPC shares the same robust login implementation and configuration used by all JUNOS login methods (ssh, telnet), but adds SSL certificates for another layer of security.

2.2.1.3 Clear-text

The clear-text access mechanism is supported to allow access in countries to which encryption technologies cannot be exported. It is completely unsecure and suffers from lack of encryption, allowing passwords to be easily sniffed off the network. It should be used as a last resort, or possibly when the entire network is considered secure, as in a lab or standalone network.

2.2.1.3.1 Direct Serial Connection

The direct serial access method is useful when the application is running on a laptop or PDA that is directly connected to the console serial port of the managed device. This scenario covers both field personnel doing chassis installation and bootstrapping, as well as catastrophic situations where console access is the only avenue left for recovery.

Using the serial access method requires the client (or its libraries) to do a small amount of expect-like work while prompting for the login name, password, and issuing the "xml-mode" command to get into XML mode. This startup dialog is similar to PPP dial-up scripts.

2.2.1.3.2 Telnet

Supporting the telnet [12] protocol is required to provide an access method of last resort for sites that cannot run SSH and SSL. Supporting telnet also gives a path for accessing console ports connected to terminal servers.

Like the direct serial connection, startup requires a small amount of expect-like logic during login prompting, similar to PPP dial-up scripts.

Support for UTF-8 [13] encoding requires the use of an 8-bit data
path, available thru "telnet -8" and the BINARY telnet option. If this option is refused by the telnet server, only the US-ASCII encoding can be supported.

Care must be taken to ensure that neither client nor server overwhelm the limited resources of the terminal server. Rate limiting is an obvious requirement.

2.3 JUNOScript Protocol

The JUNOScript protocol is ultralightweight by design. This design was driven by two major motivations. First was our desire to work over any connection-oriented transport protocol. This flexibility was important for dealing with the scenarios described above.

Second, we wanted RPCs containing XML data, not merely RPCs encoded in XML. Both XML-RPC [6] and SOAP [7] target remote invocation of traditional procedural functions, and their standard encoding rules only define treatment of simple argument data types.

In addition, XML-RPC requires the type of the encoded value to be encoded with the content, effectively mixing schema and data.

The SOAP standard does not include a mechanism for generic encoding XML, but the encodingStyle attribute can be used to reference a nonstandard encoding style:

    SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml"

Although this is supported in the Apache SOAP toolkit [23], the a priori model of encoding means that any client whose libraries don’t support this encoding style can not parse the reply.

In addition, XML-RPC and SOAP are only available over HTTP [14], which requires use of the content-length field in the HTTP header to allow a single connection to contain multiple RPCs. Constructing a content length requires buffering for the entire data payload of a reply, whose size is effectively unbounded.

The "chunked" transfer encoding was introduced in HTTP 1.1, but it was not widely supported when we began this project.

When our XML project began, no standard was in place for this sort of protocol and several XML-based RPC protocols were being considered by the W3C. Since none of them were usable in their current form, we decided to monitor future standards, but continue with the work with an RPC flavor that could easily be converted to work with any of the contenders.
With no usable standard in place, we used a lightweight RPC that could be trivially replaced with the results of a standardization effort.

BEEP [9] was announced shortly after the design phase of our project concluded, but we had no indication what its future held. BEEP matched many of our needs, and it has become more appealing as its deployment has increased. BEEP holds numerous benefits, including multiple channels (for multiple outstanding RPCs), and excellent XML support.

2.3.1 Session Protocol

A JUNOScript session consists of a series of RPCs between the client and the server. The client sends an <rpc> element and receives an <rpc-reply> element in response. The contents of each direction of the JUNOScript session form a complete XML document, allowing the entire session to be saved to an XML file for post-processing, or to be used as a component of a UNIX pipeline.

The session begins with each side sending an XML declaration and the start tag for the top-level element, <junoscript>. Each of these include a set of attributes that direct the operation of the other side. The XML declaration MUST contain the version of XML in use and SHOULD contain the character-set encoding.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<junoscript version="1.0" os="JUNOS" release="5.4R1">
```

The current version number for XML is "1.0". The encoding should be one of those defined in the IANA Charset [15] listing, which includes "US-ASCII" (7-bit ascii), "ISO-8859-1" (8-bit internationalized ascii), and "UTF-8" (8-bit unicode).

The <junoscript> start tag also includes a version attribute containing the JUNOScript session protocol, which is currently has the value "1.0". Specifying this attribute allows the peer with a higher version number to enter a backward compatible mode to communicate with the older peer. The "os" and "release" attributes are optional, and allow the same style of backwards compatibility for OS-specific issues.

The XML declaration and <junoscript> start tag form a handshake. These items should be emitted when the connection is opened. Neither side should wait for the peer to emit their part of the handshake. Emitting the handshake is considered simultaneous.

The session continues with the client sending a series of <rpc>
elements to the server and the server replying to each <rpc> with an <rpc-reply> element.

<!-- The client sends an rpc like: -->
<rpc> <!-- start tag for RPC -->
  <!-- get all interface information -->
  <get-interface-information> <!-- start tag for method -->
    <!-- arguments are encoded under the method element -->
    <statistics/> <!-- Boolean argument -->
  </get-interface-information> <!-- end tag for method -->
</rpc> <!-- end tag for RPC -->

<!-- The server replies with something like: -->
<rpc-reply> <!-- start tag for RPC reply -->
  <interface-information> <!-- start tag for contents -->
    <!-- contains a hierarchy of elements containing all interface info -->
  </interface-information> <!-- end tag for contents -->
</rpc-reply> <!-- end tag for RPC reply -->

The session ends with the end tag for the <junoscript> element:
</junoscript>

The contents of a JUNOScript session, in either direction, form a complete XML document with a top-level element called <junoscript>.

The JUNOScript server preserves all attributes given with the <rpc> start tag and returns them on the matching <rpc-reply> start tag. This provides a simple mechanism for matching <rpc-reply> elements to <rpc> elements and allows the <rpc-reply> element to carry sufficient information about the RPC to be self-contained.

<!-- the results of an rpc marked with enough information to allow self-sufficiency -->
<rpc-reply host="test3.juniper.net" date="2002/07/04" operation="get-statistics">
  <interface-information>
    <!-- content omitted -->
  </interface-information>
</rpc-reply>

2.4 Error Handling

JUNOScript’s indicator of success and failure is simply the absence or presence of the <xnm:error> element. If the RPC completes without
an error, it’s taken to mean success. Errors normally appear as the first thing returned inside the <rpc-reply> element, but some may be reported during the execution of the request. For example, if a resource limitation problem is encountered during a request for information about a hardware component, the <xnm:error> element might appear in the middle of the data returned by the <get-chassis-inventory> RPC.

The lack of a definitive success indication has been problematic and may mask certain classes of failures. We will likely be adding explicit success confirmation elements to future releases of JUNOScript.

The <xnm:error> element may contain a number of elements detailing the source and cause of the error:

```xml
<rpc-reply>
  <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm"
              xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <source-daemon>mgd</source-daemon>
    <filename>your-incoming.config</filename>
    <line-number>112</line-number>
    <column>12</column>
    <token>{</token>
    <edit-path>protocols bgp groups foo</edit-path>
    <statement>passive</statement>
    <message>syntax error: open braces unexpected</message>
  </xnm:error>
</rpc>
```

A typical error message may only contain one or two of these elements.

### 2.5 Framing Protocol Errors

When errors are detected in the framing protocol, the JUNOScript server closes the connection. Although this behavior may seem severe, the server cannot trust that it will not discard something important during error recovery that may cause the script to fail its intended purpose. Errors in RPC content are acceptable and are reported as errors, but failures of the framing protocol are fatal.

This scenario is a result of having both framing and content in XML. A distinct framing protocol would not have this problem. The choice was between a non-XML framing protocol and having the entire session form a component XML document. Given our expectation of JUNOScript as a component of UNIX pipelines in shell scripts, we chose the latter.
3. RPC Content

JUNOScript provides almost all of the same functionality as the JUNOS CLI, although there are differences between the models used by the CLI and JUNOScript. In particular, the CLI is modal, with a distinction between operational mode and configuration mode. The API is nonmodal, allowing operational and configuration commands to be interspersed throughout the session.
4. Configuration Operations

JUNOScript supports a rich set of configuration operations. Configuration data can be created, read, updated, or deleted.

JUNOS uses a commit-based configuration model. A client makes changes to a working copy of the configuration database called the "candidate configuration," without affecting the configuration currently active on the device (the "committed configuration"). To implement the changes in the candidate configuration, the client performs a commit operation. The system is informed of the incoming changes and the candidate configuration overwrites the current committed configuration. The candidate configuration also retains the changes.

4.1 Configuration Data

Configuration data can be retrieved from the device in sections of as a whole. A full configuration contains the complete configuration for the device and is all that is required to resurrect the device in case of catastrophic failure.

If a partial configuration retrieval is desired, the client can provide a hierarchy of elements specifying the portion of the configuration to be retrieved.

4.1.1 Configuration in Text Format: <configuration-text>

Configuration data can be handled in either the native JUNOS format or in XML encoding.

The native format appears inside the <configuration-text> element.
4.1.2 Configuration in XML Format: <configuration>

The JUNOS native text display algorithms contain many features to make the display more attractive to the user and to optimize the use of screen space on the CLI output terminal. When emitting XML, these considerations are replaced by a need for simplicity and consistency.

The configuration data from the previous example appears in XML as follows:

```xml
<configuration>
  <firewall>
    <filter>
      <name>test</name>
      <term>
        <name>one</name>
        <from>
          <address>
            10.1.0.0/16;
          </address>
        </from>
        <port>ssh telnet;</port>
        <then>
          <reject administratively-prohibited;/>
        </then>
      </term>
      <term>
        <name>two</name>
        <from>
          <address>
            10.1.0.0/16;
          </address>
        </from>
        <then>discard;</then>
      </term>
      <term>
        <then>accept;</then>
      </term>
    </filter>
  </firewall>
</configuration>
```
<name>10.1.0.0/16</name>
</address>
<port>ssh</port>
<port>telnet</port>
</from>
<then>
<reject>
<administratively-prohibited/>
</reject>
</then>
</term>
<term>
<name>two</name>
<from>
<address>
<name>10.1.0.0/16</name>
</address>
</from>
<then>
<discard/>
</then>
</term>
<term>
<name>three</name>
<then>
<accept/>
</then>
</term>
</filter>
</firewall>
</configuration>

There are five simple encoding rules for configuration data elements.

1. The simplest elements contain data in string form:

   <host-name>test</host-name>
   <name>phil</name>
   <interface-name>so-2/3/1</interface-name>

2. If the configuration statement is a simple Boolean or an
   enumeration, it is encoded as an empty element:

   <disable/>
   <no-readvertise/>
   <accept/>

3. The most common element is a simple container that holds other
elements. The container serves as a level of hierarchy in the configuration. Some containers have a semantic meaning even when they do not contain other elements.

For example the "ssh" element is a container and can contain items such as <root-login> and <protocol-version>, but enables the ssh protocol even when empty.

    <!-- system, services, and ssh are all containers -->
    <configuration>
        <system>
            <services>
                <ssh/>
            </services>
        </system>
    </configuration>

4. When multiple instances of the same data object exist, one or more distinguishing characteristics must be defined. For example, the name of a user distinguishing that user’s configuration data from other users.

In some cases, a free-form name must be created when no distinguishing characteristics exist. For example, a series of filter terms are given user-defined names that are used to identify the terms. The names have no operational impact but are required to allow the terms to be individually manipulated.

These distinguishing characteristics are termed "identifiers". Most list in the JUNOS configuration identify list members by a single identifier. A few statements use multiple identifiers to identify list members.

Single identifiers are encoded as the <name> element under an instance of configuration data element:
<!-- Configuration for three users -->
<configuration>
  <system> <!-- container -->
    <!-- container -->
    <login> <!-- container -->
      <!-- user instance -->
        <!-- identifiers must appear first -->
        <name>phil</name> <!-- name is an identifier -->
        <!-- a user named "phil" -->
        <!-- data for this instance -->
        <full-name>Phil Shafer</full-name>
        <class>operator</class>
      </user>
      <user> <!-- different user instance -->
        <name>rpe</name> <!-- different identifier value -->
        <full-name>Rob Enns</full-name>
        <class>support</class>
      </user>
      <user>
        <name>jim</name>
        <full-name>Jimbo Jimbo</full-name>
        <class>read-only</class>
      </user>
    </login>
  </system>
</configuration>

5. The last class of data encoding is simple sets. A simple set consists of one or more data values encoded in a series of elements.

<configuration>
  <system> <!-- one data instance: -->
    <domain-search>juniper.net</domain-search>
    <!-- a second data instance: -->
    <domain-search>englab.juniper.net</domain-search>
  </system>
</configuration>

In the native text configuration, these simple sets are displayed in square brackets:

<configuration-text>
  system {
    domain-search [ juniper.net englab.juniper.net ];
  }
</configuration-text>

If the set contains only one element, the square brackets are
optional in the text format. The XML format is unchanged.

Note that there is no mixed content in JUNOScript, where an element can contain both significant text and other elements. JUNOScript elements are either data elements or container elements. The only text that appears inside container elements is ignorable whitespace.

4.2 Retrieving Configuration with <get-configuration>

The full or partial configuration can be retrieved with the <get-configuration> RPC. The <get-configuration> element may contain any of the following attributes:

- The "database" attribute can have the value of either "candidate" or "committed", indicating which copy of the database the client wants. The default value is "candidate".

- The "format" attribute can have the value of "xml" or "text". If format has the value "xml", the server emits the <configuration> element containing configuration encoded as XML. If format has the value "text", the server emits the <configuration-text> element containing the configuration data in the native JUNOS format.

```xml
<!-- Get the full candidate configuration in XML -->
<rpc>
  <get-configuration/>
</rpc>

<!-- Get the full committed configuration in text -->
<rpc>
  <get-configuration format="text" database="committed"/>
</rpc>
```

To retrieve a subset of the configuration, the client can place a <configuration> element within the <get-configuration> element, which in turn contains the hierarchy of elements that represent the section of the configuration to retrieve:
The above RPC returns only the requested hierarchy inside a
<configuration> element:

    <rpc-reply>
      <configuration>
        <firewall>
          <filter>
            <name>test</name>
            <term>
              <name>one</name>
            </term>
            <from>
              <address>
                <name>10.1.0.0/16</name>
              </address>
              <port>ssh</port>
              <port>ssl</port>
            </from>
            <then>
              <reject>
                <administratively-prohibited/>
              </reject>
            </then>
          </term>
        </filter>
        <firewall>
      </configuration>
    </rpc-reply>
4.2.1 Implementation Note

The choice of formats for the subset specification was difficult. XPath [3] seems like a more natural fit for describing a location in the hierarchy. However using the element hierarchy gives consistent syntax between the retrieval and modification. Given a node within a DOM tree, it’s easier to generate the element hierarchy than to translate it into an XPath expression.

In addition, the costs of a full XPath implementation are larger than the simple hierarchy matching mechanism.

4.3 Pushing Configuration Data with <load-configuration>

The client can also load configuration data onto a device, specifying that the data replace the entire configuration ("load override" in CLI parlance), replace sections of the configuration ("load replace"), or augment the existing configuration ("load merge"). These options are discussed below.

The <load-configuration> RPC consists of a <load-configuration> element containing either the attribute "url", the element <configuration> or the element <configuration-text>. The "format" attribute must have the value "text" if <configuration-text> is used or may have the value "xml" if <configuration> appears. The format attribute also indicates the format of the file given by the "url" attribute. The default format is "xml".
The ability of the `<load-configuration>` RPC (and `<get-configuration>`) to handle partial data sets is a key feature, since no provider will turn their entire network over to a new application. The faith, trust, and vision to leap directly into a fully automated world effectively guarantees that progress will be gradual, and will likely be driven by deployment of new provisioning-intensive services such as VPNs [16]. Such small steps will inspire learning and confidence among the operator, tool-smith, and vendor communities. Requiring fully-managed configurations would have been an impediment to deployment and adoption.

4.3.1 Disposition of Incoming Configuration

The "action" attribute on the `<load-configuration>` element specifies
how the JUNOScript server handles the incoming data.

### 4.3.1.1 Merging Incoming Configuration Data

If "action" has the value "merge" (the default), the incoming configuration is added to the existing candidate configuration data. If only one instance of the data is allowed, the value of that data instance is set to the incoming value. If multiple values are allowed, an instance containing the incoming data is added to the configuration.

For example, the system configuration allows a single `<host-name>` element and multiple `<domain-search>` elements in the domain search path.

```xml
<!-- Example configuration -->
<configuration>
  <system>
    <host-name>test</host-name>
    <domain-search>juniper.net</domain-search>
    <domain-search>englab.juniper.net</domain-search>
  </system>
</configuration>

<!-- RPC to set the host-name -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>
        <host-name>test2</host-name>
      </system>
    </configuration>
  </load-configuration>
</rpc>

<!-- RPC to add a domain to the domain-search path -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>
        <domain-search>testlab.juniper.net</domain-search>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```
Any number of configuration changes can be performed in one <load-configuration> RPC.

<!-- RPC to set the hostname and add two new domain-search paths -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>
        <host-name>test3</host-name>
        <domain-search>poclab.juniper.net</domain-search>
        <domain-search>uklab.juniper.net</domain-search>
      </system>
      <protocols>
        <!-- additional configuration -->
      </protocols>
    </configuration>
  </load-configuration>
</rpc>

4.3.1.2 Overriding with Incoming Configuration Data

If "action" has the value "override", the incoming data completely replaces the current candidate configuration. The current candidate configuration is discarded and the incoming configuration loaded in its place.

<rpc>
  <load-configuration action="override">
    <configuration>
      <system>
        <domain-search>testlab.juniper.net</domain-search>
      </system>
    </configuration>
  </load-configuration>
</rpc>

Following this RPC, the configuration contains the one new domain-search statement under a system statement.

4.3.1.3 Partial Updates with Incoming Configuration Data

If "action" has the value "replace", handling of the incoming configuration data is contingent on the presence of the "replace" attribute on the elements under the <configuration> element.

If an element contains the "replace" attribute and it has the value
"replace", the contents of the matching statement in the candidate
configuration database are replaced with the incoming configuration.
Any element that does not have the replace attribute is treated as a
merge operation.

<!-- Replace the entire contents of the <system> hierarchy
with new configuration -->
<rpc>
    <load-configuration action="replace">
        <configuration>
            <system replace="replace">
                <host-name>test</host-name>
                <domain-search>juniper.net</domain-search>
                <domain-search>englab.juniper.net</domain-search>
            </system>
        </configuration>
    </load-configuration>
</rpc>

See Section 6.8.1 for a discussion of Boolean attribute encoding.

If the incoming configuration is in the "text" format, the replace
action looks for the string "replace:" preceding any statement and
replaces the matching statement in the candidate configuration with
the incoming data.

<rpc>
    <load-configuration action="replace" format="text">
        <configuration-text>
            replace: system {
                host-name test;
                domain-search [juniper.net englab.juniper.net ];
            }
        </configuration-text>
    </load-configuration>
</rpc>

4.4 Removing Configuration Data

A subhierarchy of configuration can be deleted by loading a
configuration element with the matching element containing the
"delete" attribute with the value "delete". The element should
contain only elements required to identify the data instance.
<!-- Delete the host-name statement -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>      <host-name delete="delete"/>      </system>    </configuration>  </load-configuration></rpc>

<!-- Delete the user named "jim" -->
<rpc>
  <load-configuration action="merge">
    <configuration>      <system>      <login>      <user delete="delete">      <name>jim</name>      </user>      </login>    </system>    </configuration>  </load-configuration></rpc>

4.5 Deactivating Configuration Data

JUNOS features a mechanism for marking a section of the configuration hierarchy as inactive. When a subhierarchy of the configuration database is inactive, it is ignored during a commit operation. The statements are effectively commented out of the configuration, but may be easily activated.

To activate a configuration hierarchy, issue a <load-configuration> RPC containing the configuration hierarchy with the attribute "active" having the value "active". To deactivate, use the attribute "inactive" with the value "inactive".
<!-- Deactivate the user "phil" -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>
        <login>
          <user inactive="inactive">
            <name>phil</name>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>

<!-- Then re-activate the user "phil" -->
<rpc>
  <load-configuration action="merge">
    <configuration>
      <system>
        <login>
          <user active="active">
            <name>phil</name>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>

4.6 Reverting to Prior Configuration Data with Rollback

JUNOS archives the ten previous configuration files on the router and supports reloading those files as a rollback operation. The CLI command "rollback 0" loads the most recently committed configuration, effectively discarding any changes made to the configuration file since the last commit, and "rollback 9" loads the oldest configuration file.

JUNOScript supports the rollback operation via the "rollback" attribute on the <load-configuration> RPC. This attribute's value should be the number of the configuration to which to roll back.
4.7 Database Locking

JUNOS supports multiuser access to the configuration database. Configuration data can be viewed and modified by multiple clients (human or script) simultaneously. To avoid disastrous interactions between clients, JUNOScript supports an exclusive lock on the configuration database. This lock is granted if two simple conditions hold:

- No uncommitted changes exist, and
- No other entity currently holds an exclusive lock

If the lock fails, a client script may retry at a later time.

The database lock does not prevent other users from entering configuration mode or retrieving configuration, but blocks any attempt to modify the candidate database or to issue a commit operation.

4.7.1 Acquiring the Exclusive Database Lock

The lock RPC is simply an empty <lock-configuration> request element inside an <rpc> element:

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

The lifespan of the lock extends until either the lock is explicitly released via the <unlock-configuration> RPC or the lock is implicitly released when the server detects that the transport connection has been closed ungracefully. Only the session currently holding the lock can release it.
If the lock is released while uncommitted changes exists, manual intervention would be required to discard (roll back) the changes or to commit them. Since this intervention is normally undesirable, JUNOScript provides a feature to automatically discard any pending changes when the lock is released. To activate this feature, set the "rollback" attribute to the value "automatic" on the <lock-configuration> element:

```xml
<rpc>
  <lock-configuration rollback="automatic"/>
</rpc>
```

The reply for this operation is either success or failure:

```xml
<!-- A successful reply (no error or confirmation) -->
<rpc-reply>
</rpc-reply>

<!-- An error indicates failure to obtain the lock -->
<rpc-reply>
  <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm"
    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <message>configuration database modified</message>
  </xnm:error>
</rpc-reply>
```

4.7.2 Releasing the Exclusive Database Lock

The lock may be explicitly released with the <unlock-configuration> RPC. After the lock is released, any user or client script is free to modify the database, commit, roll back, or acquire the exclusive lock.

The unlock RPC is an empty <unlock-configuration> element inside an <rpc> element:

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

If the "rollback" attribute had the value "automatic" on the <lock-configuration> RPC, any uncommitted changes to the database will be discarded.
4.8 Primitive Operations

The primitive operations described above provide the functionality required by each member of the operation set commonly called CRUD, for Create, Read, Update, and Delete.

- create is supported via the <load-configuration> RPC
- read is supported via the <get-configuration> RPC
- update is supported via the "replace" action and attribute, and the "merge" action, and
- delete is supported via the "delete" attribute

In addition, rollback and locking give transactional integrity to configuration changes. If changes fail, they may be completely discarded and the device restored to the state it was in before the changes were made.

4.9 Loading Configuration Patches

The CRUD operations can be combined to make configuration patches, which perform multiple changes to the candidate configuration to move it to a desired state without forcing a complete reload.

<!-- RPC to remove user "phil" and create user "pshafer" -->
<rpc>
  <load-configuration action="replace">
    <configuration>
      <system>
        <login>
          <user delete="delete">
            <name>phil</name>
          </user>
          <user replace="replace">
            <name>pshafer</name>
            <full-name>P. Shafer</full-name>
            <class>operator</class>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
4.10 Committing Configuration Data

JUNOS uses a commit-based model for configuration. When a complete set of changes is ready to be installed as the running configuration, the client commits the changes using the "commit" command. The candidate configuration is copied to the committed configuration, and the new configuration is propagated to the appropriate subsystems.

In JUNOScript, the commit operation is invoked using the <commit-configuration> RPC.

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

The configuration can be checked for syntactic and semantic errors without installing it as the committed configuration. This check only phase is invoked by placing the <check> element inside the <commit-configuration> element:

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

In addition, JUNOS supports a confirmed commit, where a second confirmation commit must occur within a timeout period. If the confirming commit is not given, the configuration is automatically rolled back to the previous configuration. If the commit causes the device or the network to fail, this automatic recovery will restore connectivity to the device when the timeout expires.

```xml
<!-- Commit with a five minute confirmation timeout -->
<rpc>
  <commit-configuration>
    <confirmed/>
    <confirm-timeout>5</confirm-timeout>
  </commit-configuration>
</rpc>
```

When a network management application commits a configuration change, the application may use a confirmed commit. The application can then run reachability tests to ensure that the change has not damaged the network. Once the stability of the network is determined, the confirming commit can be issued.

For automated configuration changes, the confirmed commit capability
is important. The ability of the network to recover from provisioning errors directly affect the network’s stability, downtime, and SLA compliance.
5. Operational Data

JUNOScript includes the ability to download operational information via simple RPCs. This ability is important for two reasons. First it is our goal to provide an API with capabilities equal to the CLI. When complete, there should be nothing that can be done in the CLI that cannot be done from the API.

Second, when writing a script that provisions the device but needs some piece of operational data, a completely distinct communications path should not be required. If an application needs to see if the device has an empty slot, it should not need to use SNMP to find out when it already has a JUNOScript session open. This ability is also important when the communications path runs over the console port. For example, an application that is doing install-time configuration loading might want to check that the device being loaded is the one that is supposed to be loaded and not some other functioning device.

5.1 Operational RPC Encoding

Operational RPCs can be encoded in two ways. Each supported operational RPC can be encoded in XML elements, with the RPC’s method name as an element inside an <rpc> element and any arguments encoded as elements within it. This encoding can be fully described with XML Schemas and DTDs for that particular OS release.

    <!-- RPC to get some bizarre routes -->
    <rpc>
        <get-route-information>
            <extensive/>
            <damping/>
            <table>inet.0</table>
            <aspath-regex>.*1.*</aspath-regex>
        </get-route-information>
    </rpc>

In addition, any JUNOS command-line text can be invoked via the <command> RPC:

    <rpc>
        <command>
            show route extensive damping
            table inet.0 aspath-regex ".*1.*"
        </command>
    </rpc>

The results of operational RPCs are encoded in <rpc-reply> elements.
<rpc-reply>
    <route-information> <!-- start tag for the results -->
        <!-- contents of the reply -->
    </route-information>
</rpc-reply>
6. XML Topics

This section discusses several XML-related topics.

6.1 Schema Validation

XML provides two main mechanisms for validation: DTDs and Schemas. DTDs lack scoping and namespace support, and are specified in a non-XML form. XML Schemas have an abundance of complexity, but are more closely aligned with the data constructs we require.

XML Schema definitions basically work as a set of constraints, restricting valid XML to follow a set of rules. These rules can include containment, order, minimums occurrence limits, maximums occurrence limits, default values, and content constraints such as data typing, range checking, pattern matching, and enumerations.

XML Schema is encoded in XML, making it possible to translate Schema into XSL transformations, XML Forms, and other XML data sets.

```xml
<!-- XML Schema for the [system login user] configuration -->
<xsd:complexType name="login-user-object">
  <xsd:sequence>
    <xsd:element name="name" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>User name (login)</xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:string"/>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="undocumented"/>
      <xsd:element ref="comment"/>
      <xsd:element name="full-name" minOccurs="0" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation>Full name</xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:choice>
    <xsd:element name="uid" minOccurs="0">
      <xsd:annotation>
      <xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Shafer & Enns Expires February 25, 2003 [Page 35]
The contents of an outgoing <rpc> or incoming <rpc-reply> can be validated against the appropriate XML schema. The validation process cannot hope to catch all errors, but it can easily avoid simple data type or encoding errors.

In addition, the <appInfo> element allows application-specific information to be placed in the schema file without disturbing the normal function of the schema. The definition for the <appInfo> element allows it to contain any elements.

We use the <appInfo> element for path reference information to allow applications to perform referential integrity checks. We also use it for encoding schema information beyond the bounds of XML Schema, including units of measure such as bytes or meters.

6.2 Per-release schema definitions

XNM provides both DTDs and Schema files for each shipped release of JUNOS. Configuration can be validated against the particular release running on the device to which it will be sent.

XML Schema information can be downloaded from the Juniper Networks web site, but can also be retrieved directly off the device using the <get-xnm-information> RPC:
<!-- Get the configuration schema -->
<rpc>
  <get-xnm-information>
    <type>xml-schema</type>
    <namespace>
      http://xml.juniper.net/junos/5.3R1/junos-configuration
    </namespace>
  </get-xnm-information>
</rpc>

The results are an <rpc-reply> element containing the appropriate xsd file, without the XML declaration.

By guaranteeing that schema definitions match operating software, JUNOScript allows tools to trust and depend on their knowledge of the data they are manipulating.

Since XML Schema files are release-specific, XML-based diff utilities can be used to expose new (and deprecated) features.

6.3 Undocumented Configuration Data

One problem with full validation is the handling of undocumented configuration statements. Any software release will contain a set of configuration statements that are not documented and are hidden from the user because they are for debugging or are not supported. If the statements are listed in the Schema files, they are suddenly exposed. If they are not listed, configuration that contains one or more undocumented statements will not validate correctly.

We solved this problem by placing undocumented statements inside an <undocumented> element:

<configuration>
  <system>
    <undocumented><dump-on-panic/></undocumented>
  </system>
</configuration>

Using this technique allows us to fully specify all documented elements in the schema file, while allowing any element to appear inside the <undocumented> element.

6.4 Comments in Configuration Data

JUNOScript encodes comments in the <junos:comment> element:
Both C-style ("/* ... */") and shell-style ("# ...") comments are supported.

These comments are distinct from XML comments, which can appear anywhere in the XML session and are discarded by the parser. Comments encoded in the <junos:comment> element are recorded in the configuration database attached to the statement that immediately follows them. Comments move when their statement moves, and are deleted when their statement is deleted.

6.5 Configuration Groups

JUNOS supports an innovative configuration mechanism called "configuration groups". Configuration groups allow sections of configuration to be collected into groups and then applied to any level of the non-group (or 'foreground') configuration. The contents of configuration groups are normally logically cohesive, but anything can appear in any configuration group.

An inheritance model is used so that configuration need only appear once in the configuration database. In contrast, a template model instantiates the template at creation time, making longer configurations and requiring substantial effort to update the configuration when the template changes. The inheritance-based model allows one change to be immediately reflected throughout the configuration database.

Configuration groups also support a pattern-matching mechanism that allows the same section of configuration to be inherited in multiple places.

Configuration entered in the foreground (outside configuration groups) always overrides data from configuration groups. You don’t need to inherit what you already have. This is a significant benefit in the area of automated configuration. If a configuration application is doing partial configurations, it can put its
configuration in a configuration group titled after the application. This segregation of configuration allows the user to see which part of the configuration is automated and which is not. It also easy to disable the automated configuration by disabling the group.

6.5.1 Operator Override

One of the classic problems in automated configuration generation is allowing the operator to override the configuration during both emergency scenarios and to supplement missing features in the provisioning system.

Having operator-specified configuration clobbered by scripts will result in discarding important information. Requiring configuration applications to understand every knob that every vendor supports is not realistic. Operators need the ability to turn trace files on to monitor recurring problems, or allow senior support personnel or vendor field support to debug connectivity issues without worrying that the locally configured parameters will be reset.

Configuration groups allow automated configuration to be overridden by statements entered manually into the foreground configuration.

Being able to circumvent the automation makes the user more accepting of the application, knowing that failures are not fatal to their network. The override mechanism is the user’s escape hatch, giving them more independence from the application and by doing so, allowing them to rely more upon it.

6.6 Whitespace Issues

Whitespace is significant in XML, but is ignored in much of JUNOScript. The only place whitespace is preserved is when it occurs inside a data element and is not the first nor last character of the content. This allows an element hierarchy to be pretty-printed (indented) or to appear with one SAX event per line. All of the following are equivalent:

```xml
<configuration>
  <system>
    <host-name>test</host-name>
  </system>
</configuration>
```
6.7 Namespace Issues

JUNOScript namespaces take the form:

    http://xml.juniper.net/<os>/<release>/<area>

The ‘os’ currently has only the value ‘junos’. The ‘area’ is the software component being described, and the release is the major, minor, and maintenance release number. For example, the 5.4R1 software releases interface component is specified under the namespace URI:

    http://xml.juniper.net/junos/5.4R1/junos-interfaces

The XML Namespace specification does not require the URI to point to anything meaningful, it just serves as a unique identifier. However for official, supported releases, we point them at the basename of the URL for the DTD, XSD, and other files (e.g. <namespace>.dtd or <namespace>.xsd).

Putting the release name in the namespace is a double-edged sword. It’s a win because it allows the namespace to be release-specific, with schema files that document the exact details of that release. The cost is that XSLT files are namespace-specific. The author of XSLT files must explicitly list the namespace of each element, and having release-specific XSLT files is painful. A mechanism can be constructed for translating template namespaces into specific ones at run time, but although this mechanism localizes the pain, it does not remove it.

6.8 Encoding Issues

JUNOScript encodes identifiers as elements inside the data instance:
These identifiers could also be encoded as attributes:

```
<configuration>
  <system>
    <login>
      <user name="phil">
        <full-name>Phil Shafer</full-name>
      </user>
    </login>
  </system>
</configuration>
```

Or all non-container elements could be encoded as attributes:

```
<configuration>
  <system>
    <login>
      <user name="phil" full-name="Phil Shafer"/>
    </login>
  </system>
</configuration>
```

These alternative encodings are equally valid, the JUNOScript encoding was chosen as part of a design decision that attributes would not be used to hold data, but would be used for modifiers. The motivation was two-fold. First, we wanted a simple mechanism to remove modifiers without having to understand them. Second, the lack of default namespace behavior on attributes would have required the presence of prefixes on all attributes.

### 6.8.1 Boolean Attributes

JUNOScript encodes Boolean attributes with the value the same as the name:
<configuration>
  <system active="active">
    <login inactive="inactive">
      <user replace="replace">
        <name>phil</name>
        <full-name>Phil Shafer</full-name>
      </user>
    </login>
    <user delete="delete">
      <name>pshafer</name>
    </user>
  </system>
</configuration>

Other common encoding are delete="true" or delete="yes", but this encoding allows for future expansion. If the "delete" attribute needs the new value "when-empty", encoding it as delete="(true|when-empty)" might be confusing.

6.9 Push-based Parsers

One feature lacking in most XML parsers is the ability to pull data. Push APIs are common, where the parser reads from a text input stream and pushes data out via callback functions. Pull parsers return data to the caller, to be processed more centrally.

Using a process-until-end-of-file parser to parse partial RPCs over a stream requires sending a virtual end-of-file indication to the parser. This is a bit awkward.

Juniper Networks provides modules that do this virtual end-of-file handling, but future efforts will likely include support for emerging pull or semi-pull parsers such as Xerces XNI.
7. Conclusion

Experience with XML-based network management protocols under JUNOS has been both fruitful and educational. JUNOScript has been well received among developers, in-house testers, customers, and third party application vendors.

Based on the experience gathered during the creation of JUNOScript, here are some suggestions for the requirements of any future XML-based configuration mechanisms.

Operate over any transport protocol:
- Use existing authentication and authorization infrastructure
- Minimize deployment costs
- Handle fallback-to-console scenarios

Require long-lived connections:
- Simplify session management
- Allow database locking
- Support commit-based model

Support partial configuration manipulation:
- Aids in staged deployment, beginning with provisioning-intensive areas
- Allows operations triggered from database changes

Create an operator override mechanism:
- Make an opt-out mechanism for application failure
- Allows temporary fixes or debugging measures

Minimize impact on the managed device:
- Minimize implementation costs
- Minimize run-time costs
- Optimize for large configuration files
While the primary intent of this memo is informational, documenting one possible solution for common configuration problems with interesting capabilities (such as configuration groups), it serves the useful purpose of establishing that a mechanism that meets the above requirements is not only possible, but has been implemented and deployed.
8. Security

JUNOScript uses the secure authentication and encryption services of the underlaying access methods SSH and SSL. Client scripts login using the same login mechanism as the CLI (passwords and pass-phrases). Remote authentication protocols such as RADIUS and TACACS+ are supported. Session are encrypted by the access methods SSH or SSL, and host authentication is provided by SSH known-hosts files or SSL certificates.
9. Legal Notices

JUNOScript, JUNOS, and Juniper Networks are trademarks of Juniper Networks, Inc.
References


[15] Internet Assigned Numbers Authority, "Official Names for


Authors’ Addresses

Philip A. Shafer
Juniper Networks
940 Main Campus Drive, Suite 100
Raleigh, NC  27606
US
EMail: phil@juniper.net

Rob Enns
Juniper Networks
1194 North Mathilda Ave
Sunnyvale, CA  94089
US
EMail: rpe@juniper.net