Network Working Group
Internet-Draft
Expires: December 18, 2004

R. Enns, Editor
Juniper Networks

June 19, 2004

# NETCONF Configuration Protocol
# draft-ietf-netconf-prot-03

## Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

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 December 18, 2004.

## Copyright Notice



## Abstract

The NETCONF configuration protocol defined in this document provides mechanisms to install, manipulate, and delete the configuration of network devices. It uses an XML-based data encoding for the configuration data as well as the protocol messages. The NETCONF protocol operations are realized on top of a simple RPC layer.

Please send comments to netconf@ops.ietf.org. To subscribe, use netconf-request@ops.ietf.org.

Table of Contents

## 1. Introduction

The NETCONF protocol defines a simple mechanism through which a network device can be managed, configuration data and system state information can be retrieved, and new configuration data can be uploaded and manipulated. The protocol allows the device to expose a full, formal, application programming interface (API). Applications can use this straight-forward API to send and receive full and partial configuration data sets.

NETCONF uses a remote procedure call (RPC) paradigm to define a formal API for the network device. A client encodes an RPC in XML [1] and sends it to a server using a secure, connection-oriented session. The server responds with a reply encoded in XML. The contents of both the request and the response are fully described in XML DTDs or XML schemas, or both, allowing both parties to recognize the syntax constraints imposed on the exchange.

A key aspect of NETCONF is that it allows the functionality of the API to closely mirror the native functionality of the device. This reduces implementation costs and allows timely access to new features. In addition, applications can access both the syntactic and semantic content of the device's native user interface.

NETCONF allows a client to discover the set of protocol extensions supported by the server. These "capabilities" permit the client to adjust its behavior to take advantage of the features exposed by the device. The capability definitions can be easily extended in a noncentralized manner. Standard and vendor-specific capabilities can be defined with semantic and syntactic rigor. Capabilities are discussed in Section 6.

The NETCONF protocol is a building block in a system of automated configuration. XML is the lingua franca of interchange, providing a flexible but fully specified encoding mechanism for hierarchical content. NETCONF can be used in concert with XML-based transformation technologies such as XSLT to provide a system for automated generation of full and partial configurations. The system can query one or more databases for data about networking topologies, links, policies, customers, and services. This data can be transformed using one or more XSLT [7] scripts from a vendor-independent data schema into a form that is specific to the vendor, product, operating system, and software release. The resulting data can be passed to the device using the NETCONF protocol.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this

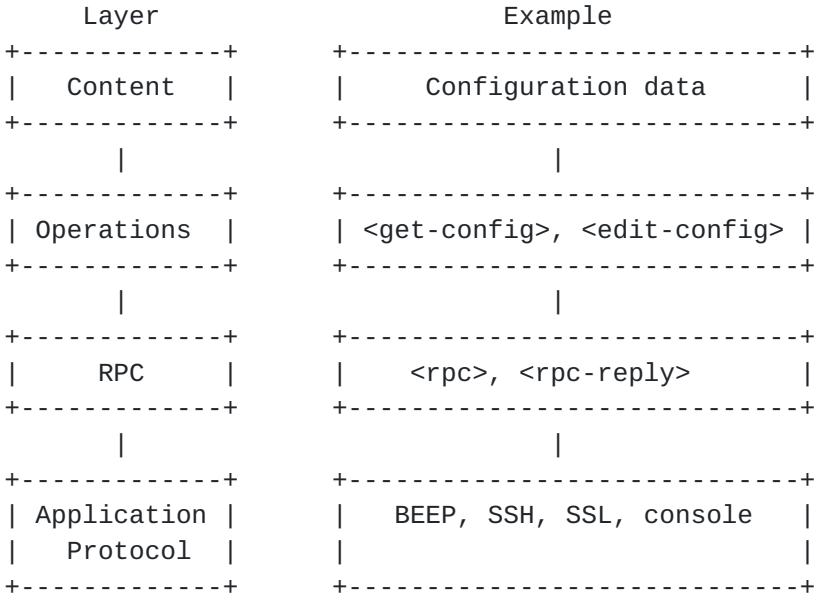
document are to be interpreted as described in RFC 2119 [2].

## 1.1 Protocol Overview

NETCONF uses a simple RPC-based mechanism to facilitate communication between a client and a server. The client is a script or application typically running as part of a network manager. The server is a network device. The terms "device" and "server" are used interchangeably in this document, as are "client" and "application".

A NETCONF session is the logical connection between a network administrator or network configuration application and a network device. A device MUST support at least one NETCONF session, and MAY support more than one. Global configuration attributes can be changed during any session, and the affects are visible in all sessions. Session-specific attributes affect only the session in which they are changed.

NETCONF can be conceptually partitioned into four layers:




1. The application protocol layer provides a communication path between the client and server. NETCONF can be layered over any application protocol that provides a set of basic requirements. Section 2 discusses these requirements.

2. The RPC layer provides a simple, transport-independent framing mechanism for encoding RPCs. Section 3 documents this protocol.

3. The operations layer defines a set of base operations invoked as RPC methods with XML-encoded parameters. Section 5 details the

list of base operations.

4. The content layer is outside the scope of this document. Given the current proprietary nature of the configuration data being manipulated, the specification of this content depends on the device vendor. It is expected that a separate effort to specify a standard data definition language and standard content will be undertaken.

## 1.2 Capabilities

A NETCONF capability is a set of functionality that supplements the base NETCONF specification. The capability is identified by a uniform resource identifier (URI). These URIs should follow the guidelines as described in Section 6.

Capabilities augment the base operations of the device, describing both additional operations and the content allowed inside operations. The client can discover the server's capabilities and use any additional operations, parameters, and content defined by those capabilities.

The capability definition may name one or more dependent capabilities. These capabilities must be implemented before the first capability can function properly. To support a capability, the server MUST support any capabilities upon which it depends.

Section 6 defines the capabilities exchange that allows the client to discover the server's capabilities. Section 6 also lists the set of capabilities defined in this document.

Additional capabilities can be defined at any time in external documents, allowing the set of capabilities to expand over time. Standards bodies may define standardized capabilities and vendors may define proprietary ones. The capability URI MUST sufficiently distinguish the naming authority to avoid naming collisions.

## 1.3 Separation of Configuration and State Data

The information that can be retrieved from a running system is separated into two classes, configuration data and state data. Configuration data is the set of writable data that is required to transform a system from its initial default state into its current state. State data is the additional data on a system that is not configuration data such as read-only status information and collected statistics. When a device is performing configuration operations a number of problems would arise if state data were included:

- Comparisons of configuration files would be dominated by irrelevant entries such as different statistics.

- A command to load the file would contain nonsensical commands such as commands to write read-only data.

- The configuration file would be too large.

To account for these issues, the NETCONF protocol recognizes the difference between configuration data and state data and provides commands that operate on each independently. The <get-config> command retrieves configuration data only while the <get> command retrieves configuration and state data.

Note that the NETCONF protocol is concerned only with information required to get the system software into its desired running state. Other important persistent data such as user files and databases are not treated as configuration data by the NETCONF protocol. Similarly, the collection of configuration files stored on a system (for example, the configuration files themselves) is not itself included in configuration data.

If a local database of user authentication data is stored on the device, whether it is included in configuration data is an implementation dependent matter.

## 2. Application Protocol Requirements

NETCONF uses an RPC-based communication paradigm. A client sends a series of one or more RPC request operations, which cause the server to respond with a corresponding series of RPC replies.

The NETCONF protocol can be layered on any application protocol that provides the required set of functionality. It is not bound to any particular application protocol, but allows a mapping to define how it can be implemented over any specific protocol.

The application protocol MUST provide a mechanism to indicate the session type (manager or agent) to the NETCONF protocol layer.

This section details the characteristics that NETCONF requires from the underlying application protocol.

### 2.1 Connection-oriented operation

NETCONF is connection-oriented, requiring a persistent connection between peers. This connection must provide reliable, sequenced data delivery.

NETCONF connections are long-lived, persisting between protocol operations. This allows the client to make changes to the state of the connection that will persist for the lifetime of the connection. For example, authentication information specified for a connection remains in effect until the connection is closed.

In addition, resources requested from the server for a particular connection MUST be automatically released when the connection closes, making failure recovery simpler and more robust. For example, when a lock is acquired by a peer, the lock persists until either explicitly released or the server determines that the connection has been terminated. If a connection is terminated while the client holds a lock, the server can perform any appropriate recovery. The lock operation is further discussed in Section 5.5

### 2.2 Security and Privacy

NETCONF connections must provide authentication, data integrity, and privacy. NETCONF depends on the application protocol for this capability. A NETCONF peer assumes that an appropriate level of security and privacy are provided independent of this document. For example, connections may be encrypted in TLS [4] or SSH [11], depending on the underlying protocol.

### 2.3 Authentication

NETCONF connections must be authenticated. The application protocol is responsible for authentication. The peer assumes that the connection's authentication information has been validated by the underlying protocol using sufficiently trustworthy mechanisms and that the peer's entity can be trusted.

One goal of NETCONF is to provide a programmatic interface to the device that closely follows the functionality of the device's native interface. Therefore, it is expected that the underlying protocol uses existing authentication mechanisms defined by the device. For example, a device that supports RADIUS [5] should use RADIUS to authenticate NETCONF sessions.

The authentication process should result in an entity whose permissions and capabilities are known to the device. These permissions must be enforced during the NETCONF session. For example, if the native user interface restricts users from changing the network interface configuration, the user should not be able to change this configuration data using NETCONF.

## 3. RPC Model

The NETCONF protocol uses an RPC-based communication model. NETCONF peers use <rpc> and <rpc-reply> elements to provide application protocol-independent framing of NETCONF requests and responses.

### 3.1 Namespace

The <rpc> and <rpc-reply> elements are defined in the following namespace:

```
urn:ietf:params:xml:ns:netconf:base:1.0
```

### 3.2 <rpc> Element

The <rpc> element is used to enclose a NETCONF request sent from the manager to the agent.

The <rpc> element has a mandatory attribute "message-id", which is an arbitrary string chosen by the sender of the RPC that will commonly encode a monotonically increasing integer. The receiver of the RPC does not decode or interpret this string but simply saves it to use as a "message-id" attribute in any resulting <rpc-reply> message. For example:

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <some-method>
    ...
  </some-method>
</rpc>
```

If additional attributes are present in an <rpc> element, a NETCONF peer must return them unmodified in the <rpc-reply> element.

The name and parameters of an RPC are encoded as the contents of the <rpc> element. The name of the RPC is an element directly inside the <rpc> element, and any parameters are encoded inside this element.

The following example invokes a method called "my-own-method" which has two parameters, "my-first-parameter", with a value of "14", and "another-parameter", with a value of "fred":

```
<rpc message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <my-own-method xmlns="http://example.net/me/my-own/1.0">
    <my-first-parameter>14</my-first-parameter>
    <another-parameter>fred</another-parameter>
  </my-own-method>
```

```
  </rpc>
```

The following example invokes a "rock-the-house" method with a "zip-code" parameter of "27606-0100":

```
  <rpc message-id="103" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <rock-the-house xmlns="http://example.net/house/rock/1.0">
      <zip-code>27606-0100</zip-code>
    </rock-the-house>
  </rpc>
```

The following example invokes the "rock-the-world" method with no parameters:

```
  <rpc message-id="104" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <rock-the-world xmlns="http://example.net/house/rock/1.0"/>
  </rpc>
```

### 3.3 <rpc-reply> Element

The <rpc-reply> message is sent in response to a <rpc> operation.

The <rpc-reply> element has a mandatory attribute "message-id", which is equal to the "message-id" attribute of the <rpc> for which this is a response.

A NETCONF peer must also return any additional attributes included in the <rpc> element unmodified in the <rpc-reply> element.

The response name and response data are encoded as the contents of the <rpc-reply> element. The name of the reply is an element directly inside the <rpc-reply> element, and any data is encoded inside this element.

For example:

```
    <rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
      <some-content>
        ...
      </some-content>
    </rpc-reply>
```

### 3.4 <rpc-error> Element

The <rpc-error> element is sent in <rpc-reply> messages if an error occurs during the processing of an <rpc> request.

The <rpc-error> element includes the following information:

error-type: Defines the conceptual layer that the error occurred. Enumeration. One of:

* transport

* rpc

* protocol

* application

error-tag: String identifying the error condition. See list below for allowed values.

error-severity: String identifying the error severity, as determined by the device. One of:

* error

* warning

error-app-tag: String identifying the data model specific or vendor specific error condition, if one exists. This element will not be present if no appropriate application error tag can be associated with a particular error condition.

error-path: Absolute XPATH expression identifying the element path to the node which is associated with the error being reported in a particular rpc-error element. This element will not be present if no appropriate payload element can be associated with a particular error condition, or if the 'bad-element' QString returned in the 'error-info' container is sufficient to identify the node associated with the error.

error-message: String describing the error condition. This element will not be present if no appropriate message is provided for a particular error condition.

error-info: Contains protocol or data model specific error content. This element will not be present if no such error content is provided for a particular error condition. The list below defines any mandatory error-info content for each error. After any protocol-mandated content, a data model definition may mandate certain application layer error information be included in the error-info container. A vendor may include additional elements at the end of the sequence to provide extended and/or

implementation-specific debugging information.

Appendix A enumerates the standard NETCONF errors.

Example: An <rpc> element is received without a message-id attribute.

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

  <rpc-reply message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
    <rpc-error>
      <error-type>rpc</error-type>
      <error-tag>MISSING_ATTRIBUTE</error-tag>
      <error-severity>error</error-severity>
      <error-info>
        <bad-attribute>message-id</bad-attribute>
        <bad-element>rpc</bad-element>
      </error-info>
    </rpc-error>
  </rpc-reply>
```

### 3.5 <ok> Element

The <ok> element is sent in <rpc-reply> messages if no error occurred during the processing of an <rpc> request. For example:

```
  <rpc-reply message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
    <ok/>
  </rpc-reply>
```

### 3.6 Pipelining

NETCONF <rpc> requests are processed serially by the managed device. Additional <rpc> requests MAY be sent before previous ones have been completed. The managed device MUST send responses only in the order the requests were received.

## 4. Configuration Model

NETCONF provides an initial set of operations and a number of capabilities that can be used to extend the base. NETCONF peers exchange device capabilities when the session is initiated as described in Section 6.1.

### 4.1 Configuration Datastores

NETCONF defines the existence of one or more configuration datastores and allows configuration operations on them. A configuration datastore is defined as the complete set of configuration data that is required to get a device from its initial default state into a desired operational state. The configuration datastore does not include state data or executive commands.

Only the <running> configuration datastore is present in the base model. Additional configuration datastores may be defined by capabilities. Such configuration datastores are available only on devices that advertise the capabilities.

- Running: The complete configuration currently active on the network device. Only one configuration datastore of this type exists on the device, and it is always present. NETCONF protocol operations refer to this datastore using the <running> element.

Section 6.3 and Section 6.7 define the <candidate> and <startup> configuration datatores, respectively.

## 5. Protocol Operations

The NETCONF protocol provides a small set of low-level operations to manage device configurations and retrieve device state information. The base protocol provides operations to retrieve, configure, copy, and delete configuration datastores. Additional operations are provided, based on the capabilities advertised by the device.

The base protocol includes the following protocol operations:

- get-config
- edit-config
- copy-config
- delete-config
- lock
- unlock
- get
- close-session
- kill-session

A protocol operation may fail for various reasons, including "operation not supported". An initiator should not assume that any operation will always succeed. The return values in any RPC reply should be checked for error responses.

The syntax and XML encoding of the protocol operations are formally defined in the XML schema in Appendix B. The following sections describe the semantics of each protocol operation.

### 5.1 <get-config>

Description:

Retrieve all or part of a specified configuration.

Parameters:

source:

Name of the configuration datastore being queried, such as <running>.

filter:

The filter element identifies the portions of the device configuration to retrieve. If this parameter is empty or unspecified, the entire configuration is returned.

[ed. May allow for different types of filters via a filter-type attribute. If so, must indicate which is mandatory to implement.]

Positive Response:

If the device can satisfy the request, the server sends an <rpc-reply> element containing a <config> element with the results of the query.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
       <rpc message-id="105" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <get-config>
           <source>
             <running/>
           </source>
           <filter>
             <config xmlns="http://example.com/schema/1.2/config">
               <users/>
             </config>
           </filter>
         </get-config>
       </rpc>

       <rpc-reply message-id="105" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <config xmlns="http://example.com/schema/1.2/config">
           <users>
             <user>
               <name>root</name>
               <type>superuser</type>
               <full-name>Charlie Root</full-name>
             </user>
             <user>
               <name>fred</name>
               <type>admin</type>
               <full-name>Fred Flintstone</full-name>
             </user>
             <user>
               <name>barney</name>
               <type>admin</type>
               <full-name>Barney Rubble</full-name>
             </user>
           </users>
         </config>
       </rpc-reply>
```

The following example shows how additional nesting within the <config> parameter can be used to filter more of the output in the response:

```
       <rpc message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <get-config>
           <source>
             <running/>
           </source>
           <filter>
             <config xmlns="http://example.com/schema/1.2/config">
               <users>
                 <user>
                   <name>fred</name>
                 </user>
               </users>
             </config>
           </filter>
         </get-config>
       </rpc>

       <rpc-reply message-id="106" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <config xmlns="http://example.com/schema/1.2/config">
           <users>
             <user>
               <name>fred</name>
               <type>admin</type>
               <full-name>Fred Flintstone</full-name>
            </user>
           </users>
         </config>
       </rpc-reply>
```

### 5.2 <edit-config>

Description:

Load all or part of a specified configuration to the specified target configuration. This operation allows the new configuration to be expressed in several ways, such as using a local file, a remote file, or inline. If the target configuration does not exist, it is created.

The device analyzes the source and target configurations and performs the requested changes. The target configuration is not simply replaced, as with the <copy-config> message.

Attributes:

operation:

Elements in the <config> subtree may contain an operation attribute. The attribute identifies the point in the configuration to perform the operation.

In the interest of simplicity, all operation attributes appearing within the <config> subtree MUST have the same value.

If the operation attribute is not specified, the configuration is merged into the configuration datastore.

The operation attribute has one of the following values:

merge: The configuration data identified by the element containing this attribute is merged with the configuration at the corresponding level in the configuration datastore identified by the target parameter.

replace: The configuration data identified by the element containing this attribute replaces any related configuration in the configuration datastore identified by the target parameter. Unlike a <copy-config> operation, which replaces the entire target configuration, only the configuration actually present in the config parameter is affected.

create: The configuration data identified by the element containing this attribute is added to the configuration if and only if the configuration data does not already exist on the device. If the configuration data exists, an <rpc-error> element is returned with an <error-tag> value of DATA_EXISTS.

delete: The configuration data identified by the element containing this attribute is deleted in the configuration datastore identified by the target parameter.

Parameters:

target:

Configuration datastore being edited, such as <running/>.

test-option:

The test-option element may be specified only if the device advertises the #validate capability (Section 6.6).

The test-option element has one of the following values:

test-then-set: Perform a validation test before attempting to set. If validation errors occur, do not perform the <edit-config> operation. This is the default test-option.

set: Perform a set without a validation test first.

error-option:

The error-option element has one of the following values:

stop-on-error: Abort the edit-config operation on first error. This is the default error-option.

ignore-error: Continue to process configuration data on error; error is recorded and negative response is generated if any errors occur.

rollback-on-error: If an error condition occurs such that an error severity <rpc-error> element is generated, the agent will stop processing the edit-config operation and restore the specified configuration to its complete state at the start of this edit-config operation. This option requires the agent to support the #rollback-on-error capability described in Section 6.5.

config:

Portion of the configuration subtree to edit. The namespace of this configuration should be specified as an attribute of this parameter.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent containing an <ok> element.

Negative Response:

An <rpc-error> response is sent if the request cannot be completed for any reason.

Example: Set the MTU to 1500 on an interface named "Ethernet0/0" in the running configuration:

```
       <rpc message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <edit-config>
           <target>
             <running/>
           </target>
           <config xmlns="http://example.com/schema/1.2/config">
             <interface>
               <name>Ethernet0/0</name>
               <mtu>1500</mtu>
             </interface>
           </config>
         </edit-config>
       </rpc>

       <rpc-reply message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
       </rpc-reply>
```

Add an interface named "Ethernet0/0" to the running configuration, replacing any previous interface with that name:

```
       <rpc message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <edit-config>
           <target>
             <running/>
           </target>
           <config xmlns="http://example.com/schema/1.2/config"
                   xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
             <interface xc:operation="replace">
               <name>Ethernet0/0</name>
               <mtu>1500</mtu>
               <address>
                 <name>1.2.3.4</name>
                 <mask>255.0.0.0</mask>
               </address>
             </interface>
           </config>
         </edit-config>
       </rpc>

       <rpc-reply message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
       </rpc-reply>
```

Delete the interface named "Ethernet0/0" from the running configuration:

```
       <rpc message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <edit-config>
           <target>
             <running/>
           </target>
           <config xmlns="http://example.com/schema/1.2/config"
                   xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
             <interface xc:operation="delete">
               <name>Ethernet0/0</name>
             </interface>
           </config>
         </edit-config>
       </rpc>

       <rpc-reply message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
       </rpc-reply>
```

Delete interface 192.168.0.1 from an OSPF area (other interfaces configured in the same area are unaffected):

```
       <rpc message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <edit-config>
           <target>
             <running/>
           </target>
           <config xmlns="http://example.com/schema/1.2/config"
                   xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
             <protocols>
               <ospf>
                 <area>
                   <name>0.0.0.0</name>
                   <interfaces>
                     <interface xc:operation="delete">
                       <name>192.168.0.1</name>
                     </interface>
                   </interfaces>
                 </area>
               </ospf>
             </protocols>
           </config>
         </edit-config>
       </rpc>

       <rpc-reply message-id="107" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
```

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

### 5.3 <copy-config>

Description:

Create or replace an entire configuration datastore with the contents of another complete configuration datastore. If the target datastore exists, it is overwritten. Otherwise, a new one is created.

A device may choose not to support the <running> configuration datastore as the <target> parameter of a <copy-config> operation. A device may choose not to support remote to remote copy operations. The source and target parameters cannot identify the same file or configuration datastore.

Parameters:

source:

The configuration datastore to use as the source of the copy operation or the <config> element containing the configuration subtree to copy.

target:

The configuration datastore to use as the destination of the copy operation.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that includes an <ok> element.

Negative Response:

An <rpc-error> element is included within the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
       <rpc message-id="108" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <copy-config>
           <source>
             <running/>
           </source>
           <target>
             <url>ftp://example.com/configs/testbed-dec10.txt</url>
           </target>
         </copy-config>
       </rpc>

       <rpc-reply message-id="108" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
       </rpc-reply>
```

### 5.4 <delete-config>

Description:

Delete a configuration datastore. The <running> configuration file cannot be deleted.

Parameters:

target:

Name of the configuration datastore to delete.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that includes an <ok> element.

Negative Response:

An <rpc-error> element is included within the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
       <rpc message-id="108" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
         <delete-config>
           <target>
             <startup/>
           </target>
         </delete-config>
       </rpc>

       <rpc-reply message-id="108" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
         <ok/>
       </rpc-reply>
```

### 5.5 <lock>

Description:

The lock operation allows the client to lock the configuration system of a device. Such locks are intended to be short-lived and allow a client to make a change without fear of interaction with other NETCONF clients, non-NETCONF clients (SNMP and Expect scripts) and human users.

An attempt to lock the configuration MUST fail if an existing session or other entity holds a lock on any portion of the lock target.

When the lock is acquired, the server MUST prevent any changes to the locked resource other than those requested by this session. SNMP and CLI requests to modify the resource MUST fail with an appropriate error.

The duration of the lock is defined as beginning when the lock is acquired and lasting until either the lock is released or the NETCONF session closes. The session closure may be explicitly performed by the client, or implicitly performed by the server based on criteria such as lack of network connectivity, failure of the underlying transport, or simple inactivity timeout. This criteria is dependent on the vendor's implementation and the underlying transport.

The lock operation takes an OPTIONAL parameter, target. If the target parameter is specified, it names the configuration that will be locked. If the target parameter is not specified, then by default the running configuration datastore will be locked. When a lock is active, using the <edit-config> operation on the locked configuration and using the locked configuration as a target of the <copy-config> operation will be disallowed by any other

session. Additionally, the system will ensure that these locked configuration resources will not be modified by other non-NETCONF management operations such as SNMP and CLI. The <kill-session> message (at the RPC layer) can be used to force the release of a lock owned by another NETCONF session. It is beyond the scope of this document to define how to break locks held by other entities.

A lock will not be granted if any of the following conditions are true:

* a lock is already held by another NETCONF session or another entity

* the target configuration has already been modified and these changes have not been committed

* lock capability not supported

The server MUST respond with either an <ok> element or an <rpc-error>.

A lock will be released by the system if the session holding the lock is terminated for any reason.

Parameters:

target:

Name of the configuration datastore to lock. If this parameter is not present, then by default the running configuration datastore will be locked.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason. This error response will include the session-id of the lock owner if the lock is already held. If the lock is held by a non-NETCONF entity, a session-id of 0 (zero) is included. Note that any other entity performing a lock on even a partial piece of a target will prevent a NETCONF lock (which is global) from being obtained on that target.

Example:

```
      <rpc message-id="120" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <lock>
          <target>
            <running/>
          </target>
        </lock>
      </rpc>

      <rpc-reply message-id="120" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
        <ok/> <!-- lock succeeded -->
      </rpc-reply>

      <rpc message-id="121" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <lock>
          <target>
            <running/>
          </target>
        </lock>
      </rpc>

      <rpc-reply message-id="121" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
        <rpc-error> <!-- lock failed -->
          <error-type>protocol</error-type>
          <error-tag>LOCK_DENIED</error-tag>
          <error-severity>error</error-severity>
          <session-id>150</session-id>
          <error-message>Lock failed, lock is already held</error-message>
          <error-info>
            <session-id>454</session-id> <!-- lock is held by NETCONF session
454 -->
          </error-info>
        </rpc-error>
      </rpc-reply>
```

### 5.6 <unlock>

Description:

The unlock operation is used to release a configuration lock, previously obtained with the <lock> operation.

An unlock operation will not succeed if any of the following conditions are true:

* the specified lock is not currently active

* the session issuing the <unlock> operation is not the same session that obtained the lock

The server MUST respond with either an <ok> element or an <rpc-error>.

Parameters:

target:

Name of the configuration datastore to unlock. If this parameter is not present, then by default the running configuration datastore will be unlocked.

A NETCONF client is not permitted to unlock a configuration datastore that it did not lock.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
<rpc message-id="121" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock>
    <target>
     <running/>
    </target>
  </unlock>
</rpc>

<rpc-reply message-id="121" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
  <ok/>
</rpc-reply>
```

### 5.7 <get>

Description:

Retrieve configuration and device state information. Appendix C describes the XML schema for NETCONF state data.

Parameters:

filter:

This parameter specifies the portion of the system configuration and state data to retrieve. If this parameter is empty, all the device configuration and state information is returned.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent. The <data> section contains the appropriate subset.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
    <rpc message-id="109" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <get>
        <filter>
          <interface xmlns="http://example.com/schema/1.2/int-stats"
name="ethernet0/1">
             <intstats/>
          </interface>
        </filter>
      </get>
    </rpc>

    <rpc-reply message-id="109" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
       <data xmlns="http://example.com/schema/1.2/int-stats">
         <interface name="ethernet0/1">
            <intstats>
               <inPkts>9456823</inPkts>
               <inOctets>1228484566</inOctets>
               <inErrors>4326</inErrors>
               <outPkts>4821050</outPkts>
               <outOctets>634712154</outOctets>
               <outErrors>2096</outErrors>
            </intstats>
         </interface>
       </data>
```

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

### 5.8 <close-session>

Description:

Request graceful termination of a NETCONF session.

When a NETCONF entity receives a <close-session> request, it will gracefully close the session. Any operations currently in process will be allowed to complete. When any in-process operations have completed, the NETCONF entity will release any locks and resources associated with the session and gracefully close any associated connections.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that includes an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
          <rpc message-id="110" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
            <close-session/>
          </rpc>

          <rpc-reply message-id="110"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
            <ok/>
          </rpc-reply>
```

### 5.9 <kill-session>

Description:

Force the termination of a NETCONF session.

When a NETCONF entity receives a <kill-session> request for an open session, it will abort any operations currently in process, release any locks and resources associated with the session and close any associated connections.

Parameters:

session-id:

Session identifier of the NETCONF session to be terminated. If this value is equal to the current session ID, a 'Bad Value' error is returned.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that includes an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
          <rpc message-id="110" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
            <kill-session>
              <session-id>4</session-id>
            </kill-session>
          </rpc>

          <rpc-reply message-id="110"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
            <ok/>
          </rpc-reply>
```

## 6. Capabilities

This section defines a set of capabilities that a client or a server MAY implement. Each peer advertises its capabilities by sending them during an initial capabilities exchange. Each peer needs to understand only those capabilities that it might use and must be able to process and ignore any capability received from the other peer that it does not require or does not understand.

Additional capabilities can be defined using the template in Appendix D. Future capability definitions may be published as standards by standards bodies or published as propriety by vendors.

A NETCONF capability is identified with a URI. The base capabilities are defined using URNs following the method described in RFC 3553 [6].

```
urn:ietf:params:netconf:base:1.0#{name}
```

Capabilities defined in this document have the following format:

```
urn:ietf:params:xml:ns:netconf:base:1.0#{name}
```

where {name} is the name of the capability. Capabilities are often referenced in discussions and email using the shorthand #{name}. For example, the foo capability would have the formal name "urn:ietf:params:xml:ns:netconf:base:1.0#foo" and be called "#foo". The shorthand form MUST NOT be used inside the protocol.

### 6.1 Capabilities Exchange

A NETCONF capability is a set of additional functionality implemented on top of the base NETCONF specification. The capability is distinguished by a URI.

Capabilities are advertised in messages sent on the NETCONF channel when each peer starts operation. When the NETCONF channel is opened, each peer sends a <hello> element containing a list of that peer's capabilities.

In the following example, the peer advertises the base NETCONF capability, one NETCONF capability defined in the base NETCONF document, and one vendor-specific capability.

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0#startup</capability>
    <capability>http:/example.net/router/2.3/core#cool-feature</capability>
  </capabilities>
</hello>
```

Each peer sends its <hello> element simultaneously as soon as the connection is open. A peer MUST NOT wait to receive the capability set from the other side before sending its own set.

## 6.2 Writable-Running Capability

### 6.2.1 Description

The #writable-running capability indicates that the device supports writes directly to the <running> configuration datastore. In other words, the device supports edit-config and copy-config operations where the <running> configuration is the target.

### 6.2.2 Dependencies

None.

### 6.2.3 Capability and Namespace

The #writable-running capability is identified by the following capability string:

```
urn:ietf:params:xml:ns:netconf:base:1.0#writable-running
```

The #writable-running capability uses the base NETCONF namespace URN.

### 6.2.4 New Operations

None.

### 6.2.5 Modifications to Existing Operations

#### 6.2.5.1 <edit-config>

The #writable-running capability modifies the <edit-config> operation to accept the <running> element as a <target>.

#### 6.2.5.2 <copy-config>

The #writable-running capability modifies the <copy-config> operation

to accept the <running> element as a <target>.

## 6.3 Candidate Configuration Capability

### 6.3.1 Description

The candidate configuration capability, #candidate, indicates that the device supports a candidate configuration datastore, which is used to hold configuration data that can manipulated without impacting the device's current configuration. The candidate configuration is a full configuration data set that serves as a work place for creating a manipulating configuration data. Additions, deletions, and changes may be made to this data to construct the desired configuration data. A <commit> operation may be performed at any time that causes the device's running configuration to be set to the value of the candidate configuration.

The candidate configuration can be used as a source or target of any operation with a <source> or <target> parameter. The <candidate> element is used to indicate the candidate configuration:

```
<rpc message-id="112" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <operation>
    <source>
      <candidate/>
    </source>
  </operation>
</rpc>
```

The candidate configuration may be shared among multiple sessions. Unless a client has specific information that the candidate configuration is not shared (for example, through another capability), it must assume that other sessions may be able to modify the candidate configuration at the same time. It is therefore prudent for a client to lock the candidate configuration before modifying it.

The client can discard any changes since the last <commit> operation by executing the <discard-changes> operation. The candidate configuration's content then reverts to the current committed configuration.

### 6.3.2 Dependencies

None.

### 6.3.3 Capability and Namespace

The #candidate capability is identified by the following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#candidate

The #candidate capability uses the base NETCONF namespace URN.

### 6.3.4 New Operations

#### 6.3.4.1 <commit>

Description:

When a candidate configuration's content is complete, the configuration data can be committed, publishing the data set to the rest of the device and requesting the device to conform to the behavior described in the new configuration.

To commit the candidate configuration as the device's new current configuration, use the <commit> operation.

The <commit> operation instructs the device to implement the configuration data contained in the candidate configuration.

If the system does not have the #candidate capability, the <commit> operation is not available.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example:

```
<rpc message-id="113" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>

<rpc-reply message-id="113" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
  <ok/>
</rpc-reply>
```

#### 6.3.4.2 <discard-changes>

If the client decides that the candidate configuration should not be committed, the <discard-changes> operation can be used to revert the candidate configuration to the current committed configuration.

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <discard-changes/>
</rpc>
```

This operation discards any uncommitted changes.

### 6.3.5 Modifications to Existing Operations

#### 6.3.5.1 <lock> and <unlock>

The candidate configuration can be locked using the <lock> operation with the <candidate> element as the <target> parameter:

```
<rpc message-id="115" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
```

Similarly, the candidate configuration is unlocked using the <candidate> element as the <target> parameter:

```
<rpc message-id="1151" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
```

On devices implementing the #candidate capability the default target of the <lock> and <unlock> operations is the candidate configuration datastore.

When a client fails with outstanding changes to the candidate configuration, recovery can be difficult. To facilitate easy recovery, any outstanding changes are discarded when the lock is released, whether explicitly with the <unlock> operation or implicitly from session failure.

#### 6.3.5.2 <get-config> and <edit-config>

The candidate configuration is the default target for the <edit-config> and <get-config> operations. It may be explicitly named using the <candidate> element:

```
<rpc message-id="117" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <candidate/>
    </source>
  </get-config>
</rpc>
```

## 6.4 Confirmed Commit Capability

### 6.4.1 Description

The #confirmed-commit capability indicates that the agent will support the <confirmed> and <confirm-timeout> parameters for the <commit> protocol operation. See section Section 6.3 for further details on the <commit> operation.

For shared configurations, this feature can cause other configuration changes (for example, via other NETCONF sessions) to be inadvertently altered or removed, unless the configuration locking feature is used (in other words, lock obtained before the edit-config operation is started). Therefore, it is strongly suggested that in order to use this feature with shared configuration databases, configuration locking must also be available and used properly.

### 6.4.2 Dependencies

The #confirmed-commit capability is only relevant if the #candidate capability is also supported.

### 6.4.3 Capability and Namespace

The #confirmed-commit capability is identified by the following capability string:

```
urn:ietf:params:xml:ns:netconf:base:1.0#confirmed-commit
```

The #confirmed-commit capability uses the base NETCONF namespace URN.

### 6.4.4 New Operations

None.

### 6.4.5 Modifications to Existing Operations

#### 6.4.5.1 <commit>

The #confirmed-commit capability allows 2 additional parameters to the <commit> operation

confirmed:

The <confirmed> element indicates that the <commit> operation MUST be reverted if a confirming commit is not issued within ten (10) minutes. The timeout period can be adjusted with the <confirm-timeout> element.

confirm-timeout:

Timeout period for confirmed commit, in minutes.

```
    <rpc message-id="114" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <commit>
        <confirmed/>
        <confirm-timeout>20</confirm-timeout>
      </commit>
    </rpc>

    <rpc-reply message-id="114" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
      <ok/>
    </rpc-reply>
```

## 6.5 Rollback on Error Capability

### 6.5.1 Description

This capability indicates that the agent will support the rollback-on-error value in the <error-option> parameter to the <edit-config> operation.

For shared configurations, this feature can cause other configuration changes (for example, via other NETCONF sessions) to be inadvertently altered or removed, unless the configuration locking feature is used (in other words, lock obtained before the edit-config operation is started). Therefore, it is strongly suggested that in order to use

this feature with shared configuration databases, configuration locking must also be available and used properly.

### 6.5.2 Dependencies

None

### 6.5.3 Capability and Namespace

The #rollback-on-error capability is identified by the following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#rollback-on-error

The #rollback-on-error capability uses the base NETCONF namespace URN.

### 6.5.4 New Operations

None.

### 6.5.5 Modifications to Existing Operations

#### 6.5.5.1 <edit-config>

The #rollback-on-error capability allows the rollback-on-error value to the <error-option> parameter on the <edit-config> operation.

```
    <rpc message-id="114" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <edit-config rollback-on-error="yes">
        <error-option>rollback-on-error</error-option>
        <target>
          <running/>
        </target>
        <config xmlns="http://example.com/schema/1.2/config">
          <interface>
            <name>Ethernet0/0</name>
            <mtu>100000</mtu>
          </interface>
        </config>
      </edit-config>
    </rpc>

    <rpc-reply message-id="114" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
      <ok/>
    </rpc-reply>
```

## 6.6 Validate Capability

### 6.6.1 Description

Validation consists of checking a candidate configuration for syntactical and semantic errors before applying the configuration to the device.

If this capability is advertised, the device supports the <validate> protocol operation and checks at least for syntax errors. In addition, this capability supports the validate parameter to the <edit-config> operation and, when it is provided, checks at least for syntax errors.

### 6.6.2 Dependencies

None.

### 6.6.3 Capability and Namespace

The #validate capability is identified by the following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#validate

The #validate capability uses the base NETCONF namespace URN.

### 6.6.4 New Operations

#### 6.6.4.1 <validate>

Description:

This protocol operation validates the contents of the specified configuration.

Parameters:

source:

Name of the configuration datastore being validated, such as <candidate>.

Positive Response:

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response:

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

A validate operation can fail for any of the following reasons:

+ Syntax errors

+ Missing parameters

+ References to undefined configuration data

Example:

```
<rpc message-id="118" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <validate>
    <candidate/>
  </validate>
</rpc>

<rpc-reply message-id="118" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
  <ok/>
</rpc-reply>
```

### 6.7 Distinct Startup Capability

#### 6.7.1 Description

The device supports separate running and startup configuration datastores. Operations which affect the running configuration will not be automatically copied to the startup configuration. An explicit <copy-config> operation from the <running> to the <startup> must be invoked to update the startup configuration to the current contents of the running configuration. NETCONF protocol operations refer to the startup datastore using the <startup> element.

#### 6.7.2 Dependencies

None.

#### 6.7.3 Capability and Namespace

The #startup capability is identified by the following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#startup

The #startup capability uses the base NETCONF namespace URN.

### 6.7.4 New Operations

None.

### 6.7.5 Modifications to Existing Operations

#### 6.7.5.1 <copy-config>

To save the startup configuration, use the copy-config command to copy the <running> configuration datastore to the <startup> configuration datastore.

```
          <rpc message-id="119" xmlns="urn:ietf:params:xml:ns:netconf:base:
1.0">
            <copy-config>
              <source>
                <running/>
              </source>
              <target>
                <startup/>
              </target>
            </copy-config>
          </rpc>
```

## 6.8 URL Capability

### 6.8.1 Description

The NETCONF peer has the ability to accept the <url> element in <source> and <target> parameters. The capability is further identified by URL arguments indicating the protocols supported.

### 6.8.2 Dependencies

None.

### 6.8.3 Capability and Namespace

The #url capability is identified by the following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#url?protocol={protocol-name,...}

The #url capability uses the base NETCONF namespace URN.

The #url capability URI MUST contain a "protocol" argument assigned a comma-separated list of protocol names indicating which protocols the

NETCONF peer supports. For example:

urn:ietf:params:xml:ns:netconf:base:1.0#url?protocol=http,ftp,file

The #url capability uses the base NETCONF namespace URN.

#### 6.8.4 New Operations

None.

#### 6.8.5 Modifications to Existing Operations

##### 6.8.5.1 <edit-config>

The #url capability modifies the <edit-config> operation to accept the <url> element as the <config> parameter.

##### 6.8.5.2 <copy-config>

The #url capability modifies the <copy-config> operation to accept the <url> element as the value of the the <source> and the <target> parameters.

##### 6.8.5.3 <delete-config>

The #url capability modifies the <delete-config> operation to accept the <url> element as the value of the the <target> parameters. If this parameter contains a URL, then it should identify a local configuration file.

##### 6.8.5.4 <validate>

The #url capability modifies the <validate> operation to accept the <url> element as the value of the the <source> parameter.

## 7. XML Usage Guidelines for NETCONF

XML serves as an encoding format for NETCONF, allowing complex hierarchical data to be expressed in a text format that can be read, saved, and manipulated with both traditional text tools and tools specific to XML.

To simplify manipulation of NETCONF content, use of XML is restricted to a simple subset of XML, as described in this section.

### 7.1 No DTDs

Document type declarations (DTDs) are not permitted to appear in NETCONF content.

### 7.2 Avoid Mixed Content

Mixed content is defined as elements that can contain both data and other elements. Elements in NETCONF can contain either data or additional elements only.

This greatly simplifies the complexity of parsing XML, especially in the area of significant whitespace. Whitespace inside data elements is significant. Whitespace outside data elements is not.

```
<valid>
  <element>data</element>
  <more>data</more>
</valid>

<not-valid>
  <element>data<more>data</more>maybe some</element>
</not-valid>
```

### 7.3 Use an Explicit Namespace on Attributes

All attributes should be prefixed so that they belong to a specific namespace. This encourages meaningful definitions that are free of collisions.

```
<valid xmlns="http://valid/" xmlns:v="http://valid/" v:foo="cool"></
valid>

<not-valid xmlns="http://not-valid/" foo="not-cool"></not-valid>
```

### 7.4 Use Container Elements for Lists

When encoding lists with multiple instances, use a distinct container element, preferably the plural form of the instance element.

In this example, the element 'grommet' is contained within the 'grommets' element.

```
    <valid>
      <grommets>
        <grommet>....</grommet>
        <grommet>....</grommet>
        <grommet>....</grommet>
      </grommets>
    </valid>
```

Use of container elements allows simpler manipulation of lists and list members.

### 7.5 Elements and Attributes

The choice of elements and attributes has been widely discussed, but no absolute guidelines exist. When designing encoding rules for NETCONF content, the following guidelines should be used:

#### 7.5.1 Consider Attributes as Metadata

Attributes should contain metadata about the element, not true data. By extension, vital information should not be encoded in attributes.

#### 7.5.2 Consider the Lack of Extensibility of Attributes

Attributes are unordered, can appear only once, and can have no children. Data scenarios which must leave room for future expansion (in future specifications or future software releases) should avoid attributes.

### 7.6 Proper Tag Names

When choosing element names, consider the following guidelines:

- Prefer ASCII (7-bit).
- Prefer lowercase.
- Prefer dashes to underscores.
- Prefer full words. Note that "config" is considered a full word.

These are guidelines only and should be considered secondary to the need for consistency with existing vocabularies. For example, when encoding MIB variables names in NETCONF, use the existing names (ifAddr) instead of shifting to these guidelines (if-address). These guidelines are valuable when no common vocabulary exists, because they help to avoid the scenario in which a dozen developers choose a dozen names that differ in ways that lead to frustrating inconsistencies, such as ifaddr, if-addr, if-address, interface-address, intf-addr, iaddr, and iface-addr.

## 8. Security Considerations

Configuration information is by its very nature sensitive. Its transmission in the clear and without integrity checking leaves devices open to classic so-called "person in the middle" attacks. Configuration information often times contains passwords, user names, service descriptions, and topological information, all of which are sensitive. Because of this, this protocol should be implemented carefully with adequate attention to all manner of attack one might expect to experience with other management interfaces.

The protocol, therefore, must minimally support options for both privacy and authentication. It is anticipated that the underlying protocol (SSH, BEEP, etc) will provide for both privacy and authentication, as is required. It is further expected that the identity of each end of a NETCONF session will be available to the other in order to determine authorization for any given request. One could also easily envision additional information such as transport and encryption methods being made available for purposes of authorization. NETCONF itself provide no means to reauthenticate, much less authenticate. All such actions occur at lower layers.

Different environments may well allow different rights prior to and then after authentication. Thus, an authorization model is not specified in this document. When an operation is not properly authorized then a simple "permission denied" is sufficient. Note that authorization information may be exchanged in the form of configuration information, which is all the more reason to ensure the security of the connection.

That having been said, it is important to recognize that some commands are clearly more sensitive by nature than others. For instance, <copy-config> to the startup or running configurations is clearly not a normal provisioning operation, where-as <edit-config> is. Similarly, just because someone says go write a configuration through the URL capability at a particular place does not mean that an element should do it without proper authorization.

The <lock> operation will demonstrate that use of NETCONF is intended for use by systems that have at least some trust of the administrator. As specified in this document, it is possible to lock portions of a configuration that a principle might not otherwise have access to. After all, the entire configuration is locked. To mitigate this problem there are two approaches. It is possible to kill another netconf session programmatically from within netconf if one knows the session identifier of the offending session. The other possible way to break a lock is to provide an function within the craft interface.

## 9. IANA Considerations

TBD.

## 10. Authors and Acknowledgements

This document was written by:

Andy Bierman, Cisco Systems

Ken Crozier, Cisco Systems

Rob Enns, Juniper Networks

Ted Goddard, IceSoft

Eliot Lear, Cisco Systems

David Perkins

Phil Shafer, Juniper Networks

Steve Waldbusser

Margaret Wasserman, ThingMagic

Normative References

[1] Bray, T., Paoli, J., Sperberg-McQueen, C. and E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C REC REC-xml-20001006, October 2000.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[3] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.

[4] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[5] Rigney, C., Willens, S., Rubens, A. and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)", RFC 2865, June 2000.

[6] Mealling, M., Masinter, L., Hardie, T. and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, June 2003.

## Informative References

[7] Clark, J., "XSL Transformations (XSLT) Version 1.0", W3C REC REC-xslt-19991116, November 1999.

[8] Hollenbeck, S., Rose, M. and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

[9] Boyer, J., "Canonical XML Version 1.0", RFC 3076, March 2001.

[10] Rose, M., "The Blocks Extensible Exchange Protocol Core", RFC 3080, March 2001.

[11] Ylonen, T., Kivinen, T., Saarinen, M., Rinne, T. and S. Lehtinen, "SSH Protocol Architecture", draft-ietf-secsh-architecture-15 (work in progress), October 2003.

## Author's Address

Rob Enns
Juniper Networks
1194 North Mathilda Ave
Sunnyvale, CA 94089
US

EMail: rpe@juniper.net

## Appendix A. NETCONF Error List

```
   Tag:         TOO_BIG
   Error-type:  transport, rpc, protocol, application
   Severity:    error
   Error-info:  none
   Description: The request or response (that would be generated) is too
                large for the implementation to handle.

   Tag:         MISSING_ATTRIBUTE
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-attribute> : name of the missing attribute
                <bad-element> : name of the element that should
                contain the missing attribute
   Description: An expected attribute is missing

   Tag:         BAD_ATTRIBUTE
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-attribute> : name of the attribute w/ bad value
                <bad-element> : name of the element that contains
                the attribute with the bad value
   Description: An attribute value is not correct; e.g., wrong type,
                out of range, pattern mismatch

   Tag:         UNKNOWN_ATTRIBUTE
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-attribute> : name of the unexpected attribute
                <bad-element> : name of the element that contains
                the unexpected attribute
   Description: An unexpected attribute is present

   Tag:         MISSING_ELEMENT
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-element> : name of the missing element
   Description: An expected element is missing

   Tag:         BAD_ELEMENT
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-element> : name of the element w/ bad value
   Description: An element value is not correct; e.g., wrong type,
                out of range, pattern mismatch

   Tag:         UNKNOWN_ELEMENT
```

```
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  <bad-element> : name of the unexpected element
   Description: An unexpected element is present

   Tag:         ACCESS_DENIED
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  none
   Description: Access to the requested RPC, protocol operation,
                or application data model is denied because
                authorization failed

   Tag:         LOCK_DENIED
   Error-type:  protocol
   Severity:    error
   Error-info:  <session-id> : session ID of session holding the
                requested lock, or zero to indicate a non-NETCONF
                entity holds the lock
   Description: Access to the requested lock is denied because the
                lock is currently held by another entity

   Tag:         RESOURCE_DENIED
   Error-type:  transport, rpc, protocol, application
   Severity:    error
   Error-info:  none
   Description: Request could not be completed because of insufficient
                resources

   Tag:         ROLLBACK_FAILED
   Error-type:  protocol, application
   Severity:    error
   Error-info:  none
   Description: Request to rollback some configuration change (via
                rollback-on-error or discard-changes operations) was
                not completed for some reason.

   Tag:         DATA_EXISTS
   Error-type:  application
   Severity:    error
   Error-info:  none
   Description: Request could not be completed because the relevant
                data model content already exists. For example,
                a 'create' operation was attempted on data which
                already exists.

   Tag:         DATA_MISSING
   Error-type:  application
```

```
   Severity:    error
   Error-info:  none
   Description: Request could not be completed because the relevant
                data model content does not exist.  For example,
                a 'modify' or 'delete' operation was attempted on
                data which does not exist.

   Tag:         OPERATION_NOT_SUPPORTED
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  none
   Description: Request could not be completed because the requested
                operation is not supported by this implementation.

   Tag:         OPERATION_FAILED
   Error-type:  rpc, protocol, application
   Severity:    error
   Error-info:  none
   Description: Request could not be completed because the requested
                operation failed for some reason not covered by
                any other error condition.


   Tag:         PARTIAL_OPERATION
   Error-type:  application
   Severity:    error
   Error-info:  <ok-element> : identifies an element in the data model
                for which the requested operation has been completed
                for that node and all its child nodes.  This element
                can appear zero or more times in the <error-info>
                container.

                <err-element> : identifies an element in the data model
                for which the requested operation has failed for that
                node and all its child nodes. This element
                can appear zero or more times in the <error-info>
                container.

                <noop-element> : identifies an element in the data model
                for which the requested operation was not attempted for
                that node and all its child nodes. This element
                can appear zero or more times in the <error-info>
                container.

   Description: Some part of the requested operation failed or was
                not attempted for some reason.  Full cleanup has
                not been performed (e.g., rollback not supported)
                by the agent.  The error-info container is used
```

to identify which portions of the application
data model content for which the requested operation
has succeeded (<ok-element>), failed (<bad-element>),
or not attempted (<noop-element>).

## Appendix B. XML Schema for NETCONF RPC and Protocol Operations

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema targetNamespace="urn:ietf:params:xml:ns:netconf:base:1.0"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
     <xs:complexType name="rpcType">
       <xs:sequence>
         <xs:element ref="rpcOperation"/>
       </xs:sequence>
       <xs:attribute name="message-id" type="xs:string" use="required"/>
     </xs:complexType>
     <xs:element name="rpc" type="rpcType"/>

     <xs:simpleType name="SessionId">
       <xs:restriction base="xs:unsignedInt"/>
     </xs:simpleType>

     <xs:simpleType name="ErrorType">
       <xs:restriction base="xs:string">
         <xs:enumeration value="transport"/>
         <xs:enumeration value="rpc"/>
         <xs:enumeration value="protocol"/>
         <xs:enumeration value="application"/>
       </xs:restriction>
     </xs:simpleType>

     <xs:simpleType name="ErrorTag">
       <xs:restriction base="xs:string">
         <xs:enumeration value="TOO_BIG"/>
         <xs:enumeration value="MISSING_ATTRIBUTE"/>
         <xs:enumeration value="BAD_ATTRIBUTE"/>
         <xs:enumeration value="UNKNOWN_ATTRIBUTE"/>
         <xs:enumeration value="MISSING_ELEMENT"/>
         <xs:enumeration value="BAD_ELEMENT"/>
         <xs:enumeration value="UNKNOWN_ELEMENT"/>
         <xs:enumeration value="ACCESS_DENIED"/>
         <xs:enumeration value="LOCK_DENIED"/>
         <xs:enumeration value="RESOURCE_DENIED"/>
         <xs:enumeration value="ROLLBACK_FAILED"/>
         <xs:enumeration value="DATA_EXISTS"/>
         <xs:enumeration value="DATA_MISSING"/>
         <xs:enumeration value="OPERATION_NOT_SUPPORTED"/>
         <xs:enumeration value="OPERATION_FAILED"/>
         <xs:enumeration value="PARTIAL_OPERATION"/>
       </xs:restriction>
```

```
    </xs:simpleType>

    <xs:simpleType name="ErrorSeverity">
```

```
      <xs:restriction base="xs:string">
        <xs:enumeration value="error"/>
        <xs:enumeration value="warning"/>
      </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="rpc-errorType">
      <xs:sequence>
        <xs:element name="error-type" type="ErrorType"/>
        <xs:element name="error-tag" type="ErrorTag"/>
        <xs:element name="error-severity" type="ErrorSeverity"/>
        <xs:element name="error-app-tag" type="xs:string" minOccurs="0"/>
        <xs:element name="error-path" type="xs:string" minOccurs="0"/>
        <xs:element name="error-message" type="xs:string" minOccurs="0"/>
        <xs:element name="error-info" type="xs:any" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>

    <!-- Standard elements used in the error-info container  -->
    <xs:element name="bad-attribute" type="xs:QName"/>
    <xs:element name="bad-element" type="xs:QName"/>
    <xs:element name="ok-element" type="xs:QName"/>
    <xs:element name="err-element" type="xs:QName"/>
    <xs:element name="noop-element" type="xs:QName"/>
    <xs:element name="session-id" type="SessionId"/>

    <xs:complexType name="rpc-replyType">
      <xs:choice>
        <xs:element name="ok" minOccurs="0"/>
        <xs:element name="rpc-error" type="rpc-errorType" minOccurs="0"/>
        <xs:element ref="config" minOccurs="0"/>
        <xs:element ref="data" minOccurs="0"/>
      </xs:choice>
      <xs:attribute name="message-id" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:element name="rpc-reply" type="rpc-replyType"/>
    <xs:simpleType name="test-optionType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="test"/>
        <xs:enumeration value="test-then-set"/>
        <xs:enumeration value="set"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="test-option" type="test-optionType"/>
    <xs:simpleType name="error-optionType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="stop-on-error"/>
        <xs:enumeration value="ignore-error"/>
```

```
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="error-option" type="error-optionType"/>
    <xs:complexType name="rpcOperationType">
      <xs:annotation>
      </xs:annotation>
    </xs:complexType>
    <xs:element name="rpcOperation" type="rpcOperationType" abstract="true"/>
    <xs:complexType name="config-inlineType">
      <xs:complexContent>
        <xs:extension base="xs:anyType">
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="config" type="config-inlineType"/>
    <xs:element name="filter" type="config-inlineType"/>
    <xs:complexType name="config-nameType"/>
    <xs:element name="config-name" type="config-nameType" abstract="true"/>
    <xs:element name="startup" type="config-nameType"
substitutionGroup="config-name"/>
    <xs:element name="candidate" type="config-nameType"
substitutionGroup="config-name"/>
    <xs:element name="running" type="config-nameType"
substitutionGroup="config-name"/>
    <xs:complexType name="config-uriType">
      <xs:simpleContent>
        <xs:extension base="xs:anyURI">
         </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
    <xs:element name="url" type="config-uriType"/>
    <xs:complexType name="rpcOperationSourceType">
      <xs:choice>
        <xs:element ref="config"/>
        <xs:element ref="config-name"/>
        <xs:element ref="url"/>
      </xs:choice>
    </xs:complexType>
    <xs:element name="source" type="rpcOperationSourceType"/>
    <xs:complexType name="rpcOperationTargetType">
      <xs:choice>
        <xs:element ref="config-name"/>
        <xs:element ref="url"/>
      </xs:choice>
    </xs:complexType>
    <xs:element name="target" type="rpcOperationTargetType"/>
    <xs:complexType name="get-configType">
      <xs:complexContent>
```

```
          <xs:extension base="rpcOperationType">
            <xs:sequence>
              <xs:element ref="source"/>
```

```
            <xs:element ref="filter" minOccurs="0"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="get-config" type="get-configType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="edit-configType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="source" minOccurs="0"/>
            <xs:element ref="target"/>
            <xs:element ref="test-option" minOccurs="0"/>
            <xs:element ref="error-option" minOccurs="0"/>
            <xs:element ref="config" minOccurs="0"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="edit-config" type="edit-configType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="copy-configType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="source"/>
            <xs:element ref="target"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="copy-config" type="copy-configType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="delete-configType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="target"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="delete-config" type="delete-configType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="getType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
```

```
            <xs:sequence>
              <xs:element ref="filter" minOccurs="0"/>
            </xs:sequence>
          </xs:extension>
```

```
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="get" type="getType" substitutionGroup="rpcOperation"/>
    <xs:complexType name="lockType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="target"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="lock" type="lockType" substitutionGroup="rpcOperation"/>
    <xs:complexType name="unlockType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="target"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="unlock" type="unlockType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="validateType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="source"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="validate" type="validateType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="commitType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element name="confirmed" minOccurs="0"/>
            <xs:element name="confirmed-timeout" minOccurs="0"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="commit" type="commitType"
substitutionGroup="rpcOperation"/>
    <xs:complexType name="discard-changesType">
```

```
      <xs:complexContent>
        <xs:extension base="rpcOperationType"/>
      </xs:complexContent>
```

```
     </xs:complexType>
     <xs:element name="discard-changes" type="discard-changesType"
substitutionGroup="rpcOperation"/>
     <xs:complexType name="close-sessionType">
       <xs:complexContent>
         <xs:extension base="rpcOperationType"/>
       </xs:complexContent>
     </xs:complexType>
     <xs:element name="close-session" type="close-sessionType"
substitutionGroup="rpcOperation"/>
     <xs:complexType name="kill-sessionType">
       <xs:complexContent>
         <xs:extension base="rpcOperationType">
           <xs:sequence>
             <xs:element name="session-id" type="SessionId" minOccurs="0"/>
           </xs:sequence>
         </xs:extension>
       </xs:complexContent>
     </xs:complexType>
     <xs:element name="kill-session" type="kill-sessionType"
substitutionGroup="rpcOperation"/>
   </xs:schema>
```

## Appendix C. XML Schema for NETCONF State Data

```
    <schema
        targetNamespace="http://ietf.org/netconf/1.0/state"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xc="http://ietf.org/netconf/1.0/state"
        elementFormDefault="unqualified">

     <annotation>
      <documentation xml:lang="en">
       Initial schema for NETCONF state information.
      </documentation>
     </annotation>

     <element name="netconf-state">
      <complexType>
       <sequence>

        <element name="capabilities">
         <annotation>
          <documentation xml:lang="en">
           List of NETCONF capabilities supported by this device.
          </documentation>
         </annotation>
         <complexType>
          <sequence>
           <element name="capability" type="anyURI"
            minOccurs="0" maxOccurs="unbounded"/>
          </sequence>
         </complexType>
        </element>

        <element name="sessions">
         <annotation>
          <documentation xml:lang="en">
           List of NETCONF sessions currently active on this device.
          </documentation>
         </annotation>
         <complexType>
          <sequence>
           <element name="my-session-id" type="positiveInteger"/>
           <element name="session" type="xc:NetconfSessionInfo"
            minOccurs="0" maxOccurs="unbounded"/>
          </sequence>
         </complexType>
        </element>
```

```
        <element name="configs">
         <annotation>
          <documentation xml:lang="en">
           List of NETCONF configuration databases supported on this device.
          </documentation>
         </annotation>
         <complexType>
          <sequence>
           <element name="config" type="xc:NetconfConfigInfo"
            minOccurs="0" maxOccurs="unbounded"/>
          </sequence>
         </complexType>
        </element>

      </sequence>
     </complexType>
    </element>

    <complexType name="NetconfSessionInfo">
     <sequence>
      <element name="session-id" type="positiveInteger"/>
      <element name="username" type="string"/>
      <element name="login-time" type="dateTime"/>
     </sequence>
    </complexType>

    <complexType name="NetconfConfigInfo">
     <sequence>
      <element name="config-name" type="xc:ConfigName"/>
      <element name="lock-status" type="xc:LockStatus"/>
     </sequence>
    </complexType>

    <complexType name="ConfigName">
     <choice>
      <element name="candidate"/>
      <element name="running"/>
      <element name="startup"/>
     </choice>
    </complexType>

    <complexType name="LockStatus">
     <sequence>
      <element name="lock-state">
       <simpleType>
        <restriction base="string">
         <enumeration value="locked"/>
         <enumeration value="unlocked"/>
```

```
         </restriction>
        </simpleType>
       </element>
       <element name="locked-by" type="positiveInteger"
        minOccurs="0"/>
      </sequence>
     </complexType>

    </schema>
```

# Appendix D. Capability Template

## D.1 capability-name (template)

### D.1.1 Overview

### D.1.2 Dependencies

### D.1.3 Capability and Namespace

The {name} is identified by following capability string:

urn:ietf:params:xml:ns:netconf:base:1.0#{name}

The {name} capability uses the base NETCONF namespace URN.

### D.1.4 New Operations

#### D.1.4.1 <op-name>

### D.1.5 Modifications to Existing Operations

#### D.1.5.1 <op-name>

If existing operations are not modified by this capability, this section may be omitted.

### D.1.6 Interactions with Other Capabilities

If this capability does not interact with other capabilities, this section may be omitted.

# Appendix E. Configuring Multiple Devices with NETCONF

## E.1 Operations on Individual Devices

Consider the work involved in performing a configuration update against a single individual device. In making a change to the configuration, the application needs to build trust that its change has been made correctly and that it has not impacted the operation of the device. The application (and the application user) should feel confident that their change has not damaged the network.

Protecting each individual device consists of a number of steps:

- Acquiring the configuration lock.
- Loading the update.
- Validating the incoming configuration.
- Checkpointing the running configuration.
- Changing the running configuration.
- Testing the new configuration.
- Making the change permanent (if desired).
- Releasing the configuration lock.

Let's look at the details of each step.

### E.1.1 Acquiring the Configuration Lock

A lock should be acquired to prevent simultaneous updates from multiple sources. If multiple sources are affecting the device, the application is hampered in both testing of its change to the configuration and in recovery should the update fail. Acquiring a short-lived lock is a simple defense to prevent other parties from introducing unrelated changes while.

The lock can be acquired using the <lock> operation.

```
<rpc message-id="201" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <lock>
    <target>
      <running/>
    </target>
  </lock>
```

```
    </rpc>
```

### E.1.2 Loading the Update

The configuration can be loaded onto the device without impacting the running system. If the #url capability is supported, incoming changes can be placed in a local file.

```
   <rpc message-id="203" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
     <copy-config>
       <source>
         <config>
           <!-- place incoming configuration here -->
         </config>
       </source>
       <target>
         <url>file://incoming.conf</url>
       </target>
     </copy-config>
   </rpc>
```

If the #candidate capability is supported, the candidate configuration can be used.

```
   <rpc message-id="204" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
     <edit-config>
       <source>
         <config>
           <!-- place incoming configuration here -->
         </config>
       </source>
       <target>
         <candidate/>
       </target>
     </edit-config>
   </rpc>
```

If the update fails, the user file can be deleted using the <delete-config> operation or the candidate configuration reverted using the <discard-changes> operation.

### E.1.3 Validating the Incoming Configuration

Before applying the incoming configuration, it is often useful to validate it. Validation allows the application to gain confidence that the change will succeed and simplifies recovery if it does not.

If the device supports the #url capability, use the <validate> operation with the <source> parameter set to the proper user file:

```
<rpc message-id="205" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <validate>
    <source>
      <url>file://incoming.conf</url>
    </source>
  </validate>
</rpc>
```

If the device supports the #candidate capability, some validation will be performed as part of loading the incoming configuration into the candidate. For full validation, either pass the <validate> parameter during the <edit-config> step given above, or use the <validate> operation with the <source> parameter set to <candidate>.

```
<rpc message-id="206" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <validate>
    <source>
      <candidate/>
    </source>
  </validate>
</rpc>
```

### E.1.4 Checkpointing the Running Configuration

The running configuration can be saved into a local file as a checkpoint before loading the new configuration. If the update fails, the configuration can be restored by reloading the checkpoint file.

The checkpoint file can be created using the <copy-config> operation.

```
<rpc message-id="207" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <copy-config>
    <source>
      <running/>
    </source>
    <target>
      <url>file://checkpoint.conf</url>
    </target>
  </copy-config>
</rpc>
```

To restore the checkpoint file, reverse the source and target parameters.

### E.1.5 Changing the Running Configuration

When the incoming configuration has been safely loaded onto the device and validated, it is ready to impact the running system.

If the device supports the #url capability, use the <edit-config> operation to merge the incoming configuration into the running configuration.

```
<rpc message-id="208" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <source>
      <url>file://incoming.conf</url>
    </source>
    <target>
      <running/>
    </target>
  </edit-config>
</rpc>
```

If the device supports the #candidate capability, use the <commit> operation to set the running configuration to the candidate configuration. Use the <confirmed> parameter to allow automatic reverting to the original configuration if connectivity to the device fails.

```
<rpc message-id="209" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit>
    <confirmed/>
    <confirm-timeout>15</confirm-timeout>
  </commit>
</rpc>
```

### E.1.6 Testing the New Configuration

Now that the incoming configuration has been integrated into the running configuration, the application needs to gain trust that the change has affected the device in the way intended without affecting it negatively.

To gain this confidence, the application can run tests of the operational state of the device. The nature of the test is dependent on the nature of the change and is outside the scope of this document. Such tests may include reachability from the system running the application (using ping), changes in reachability to the rest of the network (by comparing the device's routing table), or inspection of the particular change (looking for operational evidence

of the BGP peer that was just added).

### E.1.7 Making the Change Permanent

When the configuration change is in place and the application has sufficient faith in the proper function of this change, the application should make the change permanent.

If the device supports the #startup capability, the current configuration can be saved to the startup configuration by using the startup configuration as the target of the <copy-config> operation.

```
<rpc message-id="210" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <copy-config>
    <source>
      <running/>
    </source>
    <target>
      <startup/>
    </target>
  </copy-config>
</rpc>
```

If the device supports the #candidate capability and a confirmed commit was requested, the confirming commit must be sent before the timeout expires.

```
<rpc message-id="211" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>
```

### E.1.8 Releasing the Configuration Lock

When the configuration update is complete, the lock must be released, allowing other applications access to the configuration.

Use the <unlock> operation to release the configuration lock.

```
<rpc message-id="212" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock/>
</rpc>
```

## E.2 Operations on Multiple Devices

When a configuration change requires updates across a number of devices, care should be taken to provide the required transaction

semantics. The NETCONF protocol contains sufficient primitives upon which transaction-oriented operations can be built. Providing complete transactional semantics across multiple devices is prohibitively expensive, but the size and number of windows for failure scenarios can be reduced.

There are two classes of multidevice operations. The first class of allows the operation to fail on individual devices without requiring all devices to revert to their original state. The operation can be retried at a later time, or its failure simply reported to the user. A example of this class might be adding an NTP server. For this class of operations, failure avoidance and recovery are focused on the individual device. This means recovery of the device, reporting the failure, and perhaps scheduling another attempt.

The second class is more interesting, requiring that the operation should complete on all devices or be fully reversed. The network should either be transformed into a new state or be reset to its original state. For example, a change to a VPN may require updates to a number of devices. Another example of this might be adding a class-of-service definition. Leaving the network in a state where only a portion of the devices have been updated with the new definition will lead to future failures when the definition is referenced.

To give transactional semantics, the same steps used in single device operations listed above are used, but are performed in parallel across all devices. Configuration locks should be acquired on all target devices and kept until all devices are updated and the changes made permanent. Configuration changes should be uploaded and validation performed across all devices. Checkpoints should be made on each device. Then the running configuration can be changed, tested, and made permanent. If any of these steps fail, the previous configurations can be restored on any devices upon which it was changed. After the changes have been completely implemented or completely discarded, the locks on each device can be released.

## Appendix F. Change Log

RFC Editor: Please remove this section before RFC publication.

### F.1 draft-ietf-netconf-prot-03

Refer to the NETCONF issue list for futher detail on the issue numbers below. The issue list is found at http://www.nextbeacon.com/netconf/.

- Consistent naming of <confirm-timeout> element.
- Add #confirmed-commit capability (issue 10.3.2)
- Use a URN for the NETCONF namespace (issue 11.1.2) and capabilities
- Remove #manager capability (issue 11.2.1)
- Remove #agent capability (issue 11.2.2)
- Add "create" as a value for the operation attribute in <edit-config> (issue 13.3.1)
- Add #rollback-on-error capability (issue 13.3.2)
- Rename <get-all> operation to <get>.
- Remove format parameter from two <get-config> and one <get> examples missed in the -02 draft (issue 13.3.3).
- Add text indicating that the session-id is returned if the lock is already held (issue 13.12.3). Add example of this.
- Remove <discard-changes> parameter on the <lock> operation (issue 13.16.1), all outstanding changes are to be discarded when the candidate configuration is unlocked.
- Remove section 8.7, guidelines on namespace construction.
- Add clarifying text regarding locks held by other entities.
- Update the abstract.
- Remove mention of the format parameter from the <get-config> and <get> operations and the XSD.
- Updated security considerations section.

- Removed terminology section, moved session description to protocol overview section.
- New text describing <rpc-error>.
- Updated NETCONF protocol schema (to reflect new <rpc-error> details, among other things).
- Add <filter> parameter to <get> and <get-config>. Rename <state> response the <get> operation to <data>.
- Better description of the <kill-session> operation.
- Add <close-session> operation.
- Removed format parameter to <copy-config>.
- Removed restriction that a changed <candidate/> configuration datastore can't be locked.
- Add note in section 2 that the application protocol must provide an indication of session type (manager or agent) to the NETCONF layer.

### F.2 draft-ietf-netconf-prot-02

Refer to the NETCONF issue list for futher detail on the issue numbers below. The issue list is found at http://www.nextbeacon.com/netconf/.

- Remove <rpc-abort>, <rpc-abort-reply>, and <rpc-progress> (issues 12.1, 12.2, 12.3).
- Remove channels (issues 3.*).
- Remove notifications (issues 2.*, 4.2, 13.9, 13.10, 13.11).
- Move version number to last component of the capability URI (issue 11.1.1).
- Remove format parameter from <get-config> (issue 13.3.3).
- Remove mention of #lock capability from Appendix E. Locking is a mandatory NETCONF operation.
- Added text indicating that attributes received in <rpc> should be echoed on <rpc-reply> (issue 16.1).

- Reworded Section 7.3 to encourage always prefixing attributes with namespaces.

## Intellectual Property Statement



## Full Copyright Statement

## Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.