Network Working Group
Internet-Draft
Expires: August 19, 2005

R. Enns, Ed.
Juniper Networks

February 18, 2005

NETCONF Configuration Protocol
draft-ietf-netconf-prot-05

Status of this Memo

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 Extensible Markup Language (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 Remote Procedure Call (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 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 a 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 non-standard capabilities can be defined with semantic and syntactic rigor. Capabilities are discussed in Section 8.

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 [9] 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 scripts from a task-oriented, 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

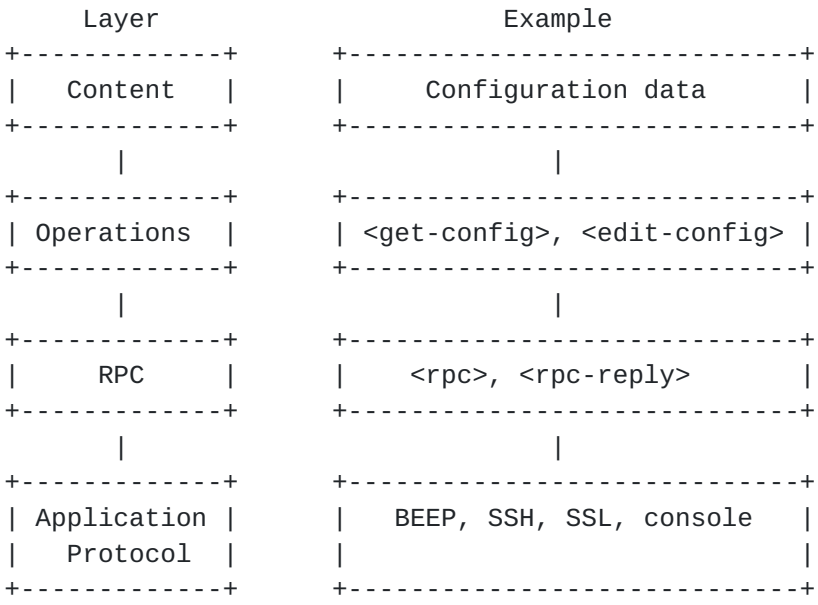
document are to be interpreted as described in RFC 2119 [3].

1.1 Protocol Overview

NETCONF uses a simple RPC-based mechanism to facilitate communication between a client and a server. The client can be a script or application typically running as part of a network manager. The server is typically 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 SHOULD support multiple sessions. Global configuration attributes can be changed during any authorized 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 4 documents this protocol.

3. The operations layer defines a set of base operations invoked as

RPC methods with XML-encoded parameters. Section 7 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 NETCONF implementation. 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 8.

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. To support a capability, the server MUST support any capabilities upon which it depends.

Section 8 defines the capabilities exchange that allows the client to discover the server's capabilities. Section 8 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 implementations may define proprietary ones. A 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 data sets would be dominated by irrelevant entries such as different statistics.
- Incoming data could contain nonsensical requests, such as attempts to write read-only data.
- The data sets would be large.
- Archived data could contain values for read-only data items, complicating the processing required to restore archived data.

To account for these issues, the NETCONF protocol recognizes the difference between configuration data and state data and provides operations for each. The <get-config> operation retrieves configuration data only while the <get> operation retrieves configuration and state data.

Note that the NETCONF protocol is focused on the information required to get the device into its desired running state. The inclusion of other important, persistent data is implementation specific. For example, user files and databases are not treated as configuration data by the NETCONF protocol.

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 (client or server) 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 client, 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 7.5.

2.2 Authentication, Integrity, 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 [10], 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 identity has been sufficiently proven.

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 [7] should allow the use of RADIUS to authenticate NETCONF sessions.

The authentication process should result in an identity whose permissions are known to the device. These permissions MUST be enforced during the remainder of the NETCONF session.

3. XML Considerations

XML serves as the 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.

This section discusses a small number of XML-related considerations pertaining to NETCONF.

3.1 Namespace

All NETCONF protocol elements are defined in the following namespace:

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

Capability names may be either URIs [5] or URNs [6].

3.2 No DTDs

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

4. 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.

4.1 <rpc> Element

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

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="101"
       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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <rock-the-house xmlns="http://example.net/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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <rock-the-world xmlns="http://example.net/rock/1.0"/>
  </rpc>
```

4.2 <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>
    <!-- contents here... -->
  </some-content>
</rpc-reply>
```

4.3 <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: Contains a string identifying the error condition. See Appendix A for allowed values.

error-severity: Contains a string identifying the error severity, as determined by the device. One of:

* error
* warning

error-app-tag: Contains a string identifying the data model specific or implementation 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: Contains the absolute XPath [2] 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: Contains a string suitable for human display which describes 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 in Appendix A 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. An implementation may include additional elements to provide extended and/or implementation-specific debugging information.

Appendix A enumerates the standard NETCONF errors.

Example:

An error is returned if an <rpc> element is received without a message-id attribute. Note that only in this case is it acceptable for the NETCONF peer to omit the message-id attribute in the <rpc-reply> element.

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

<rpc-reply 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>
```

4.4 <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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

4.5 Pipelining

NETCONF <rpc> requests MUST be 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.

5. 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 8.1.

5.1 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.

The capabilities in Section 8.3 and Section 8.7 define the <candidate> and <startup> configuration datastores, respectively.

6. Subtree Filtering

6.1 Overview

XML subtree filtering is a mechanism that allows an application to select particular XML subtrees to include in the <rpc-reply> for a <get> or <get-config> operation. A small set of filters for inclusion, simple content exact-match, and selection is provided, which allows some useful, but also very limited selection mechanisms. The server does not need to utilize any data-model specific semantics during processing, allowing for simple and centralized implementation strategies.

A subtree filter is comprised of well-formed XML. No special tags, content, or structure are imposed in any way. It is possible that a subtree filter expression may contain an empty leaf node, even if the schema for the particular data model indicates some content is required (i.e., maxOccurs > 0). For this reason, it may not be possible to completely validate all filter expressions against a schema which represents the fully populated data model.

Conceptually, a subtree filter is comprised of zero or more element subtrees, which represent the filter selection criteria. At each containment level within a subtree, the set of sibling nodes is logically processed by the server to determine if its subtree (and path to the root) are included in the filter output.

Element nodes have different purposes, depending on their position in the subtree. Most nodes in the subtree identify containment, in which the specified node name must exactly match a corresponding node in the server's data model. A leaf node in the filter with simple content is used to select matching data (some or all of the set of sibling nodes which includes this leaf node).

XML attributes can be present in any node in the filter. Each attribute acts as an additional term for the "AND" expression for that particular node. In addition to matching the name and position of the element, only instances matching all specified attribute values will be included in the response.

XML namespaces may be specified (via 'xmlns' declarations) within the filter data model. If so, the declared namespace must first exactly match a namespace supported by the server. Only data associated with a specified namespace will be considered in the filter operation.

Response messages contain only the subtrees selected by the filter. Any selection criteria that were present in the request, within a particular selected subtree, is also included in the response. Specific data instances are not duplicated in the response in the event the request contains multiple filter subtree expressions which select the same data.

6.2 Subtree filter components

A subtree filter is comprised of XML elements and their XML attributes. An attribute which appears in a subtree filter is called an "attribute match expression". All elements present in a particular subtree within a filter must match associated nodes present in the server's conceptual data model.

Nodes which contain only child elements are called "containment nodes". Leaf nodes which contain simple content are called "content match nodes". Empty leaf nodes are called "selection nodes".

6.3 Attribute Match Expressions

Any number of (unqualified or qualified) XML attributes may be present in any type of filter node. In addition to the selection criteria normally applicable to that node, the selected data must have matching values for every attribute specified in the node. If an element is not defined to include a specified attribute, then it is not selected in the filter output.

6.4 Containment Nodes

Each node specified in a subtree filter represents an inclusive filter. Only associated nodes in the specified configuration datastore on the server are selected by the filter. A node must exactly match the namespace and absolute path name of the filter data, except the filter absolute path name is adjusted to start from the layer below <filter>. A containment node must have one or more child nodes.

6.5 Content Match Nodes

A leaf node with simple content represents an exact-match filter on the simple content, which is combined (as an "AND" expression") with the criteria for a containment node, to select data model instances. All sibling content match nodes combine to represent a logical "AND" expression.

If all specified sibling content match nodes in a subtree filter expression are 'true', then the server determines the nodes to be selected in the following manner:

- If the set of all sibling nodes (at a given processing level) includes content match nodes, then all sibling subtrees (including the content match nodes) are selected by the filter.

- If the set of all sibling nodes (at a given processing level) includes any containment or selection nodes, then the content match nodes, plus any sibling subtrees selected by further processing, are selected by the filter.

- If a containment node is present in the sibling set, then it is potentially selected. If further recursive processing of its child nodes produces any selected filter output, then the containment nodes of the element hierarchy related to the particular subtree are included in the filter results.

- If a selection node is present in the sibling set, then its subtree is selected in the filter output.

If any of the sibling content match node tests are 'false', then no further filter processing is performed on that sibling set, and none of the sibling subtrees are selected by the filter, including the content match node(s).

Note that explicit selection of empty content by specifying an empty content match node is not supported.

6.6 Selection Nodes

An empty leaf node represents an "explicit selection" filter. Presence of any selection nodes (within a set of sibling nodes) will cause the filter to select the specified subtree(s). If a particular sibling set contains content match nodes and selection or containment nodes, then automatic selection of the entire sibling subtree is suppressed, and only the specified selection and/or containment nodes are selected by the filter. Unless further restricted by the application of attribute match expressions or content match nodes in the entire element hierarchy containing a particular node, the presence of a selection node will cause all instances of that node (and each node's subtree) to be selected by the filter output.

6.7 Subtree Filter Processing

The filter output (the set of selected nodes) is initially empty.

Each subtree filter can contain one or more data model fragments, which represent portions of the data model which should be selected (with all child nodes) in the filter output.

Each subtree data fragment is compared by the server to the internal data models supported by the server. If the entire subtree data-fragment filter (starting from the root to the innermost element specified in the filter) exactly matches a corresponding portion of the supported data model, then that node and all its children are included in the result data.

The server processes all nodes with the same parent node (sibling set) together, starting from the root to the leaf nodes. The root elements in the filter are considered to be in the same sibling set (assuming they are in the same namespace), even though they do not have a common parent.

For each sibling set, the server determines which nodes are included (or potentially included) in the filter output, and which sibling subtrees are excluded (pruned) from the filter output. The server first determines which types of nodes are present in the sibling set, and processes the nodes according to the rules for their type. If any nodes in the sibling set are selected, then the process is recursively applied to the sibling sets of each selected node. The algorithm continues until all sibling sets in all subtrees specified in the filter have been processed.

6.8 Subtree Filtering Examples

6.8.1 No filter

Leaving out the filter on the get operation returns the entire data model.

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

  <rpc-reply message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
      <!-- ... entire set of data returned ... -->
    </data>
  </rpc-reply>
```

6.8.2 Empty filter

An empty filter will select nothing because no content match or selection nodes are present. This is not an error. The filter type attribute used in these examples is discussed further in Section 7.1.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get>
      <filter type="subtree">
      </filter>
    </get>
  </rpc>

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

6.8.3 Select the entire <users> subtree

This filter in this example contains one selection node (<users>), so just that subtree is selected by the filter. This example represents the fully-populated <users> data model in most of the filter examples that follow. In a real data model, the 'company-info' would not likely be returned with the list of users for a particular host or network.

NOTE: The filtering and configuration examples used in this document appear in the namespace "http://example.com/schema/1.2/config". The root element of this namespace is <top>. The <top> element and it's descendents represent an example configuration data model only.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-config>
      <source>
        <running/>
      </source>
      <filter type="subtree">
        <top xmlns="http://example.com/schema/1.2/config">
          <users/>
        </top>
      </filter>
    </get-config>
  </rpc>

  <rpc-reply message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
      <top xmlns="http://example.com/schema/1.2/config">
        <users>
          <user>
            <name>root</name>
            <type>superuser</type>
            <full-name>Charlie Root</full-name>
            <company-info>
              <dept>1</dept>
              <id>1</id>
            </company-info>
          </user>
          <user>
            <name>fred</name>
            <type>admin</type>
            <full-name>Fred Flintstone</full-name>
            <company-info>
              <dept>2</dept>
              <id>2</id>
            </company-info>
          </user>
          <user>
            <name>barney</name>
            <type>admin</type>
            <full-name>Barney Rubble</full-name>
            <company-info>
              <dept>2</dept>
```

```
              <id>3</id>
            </company-info>
          </user>
        </users>
      </top>
    </data>
  </rpc-reply>
```

The following filter request would have produced the same result, but only because the container <users> defines one child element (<user>)

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-config>
      <source>
        <running/>
      </source>
      <filter type="subtree">
        <top xmlns="http://example.com/schema/1.2/config">
          <users>
            <user/>
          </users>
        </top>
      </filter>
    </get-config>
  </rpc>
```

6.8.4 Select all <name> elements within the <users> subtree

This filter contains two containment nodes (<users>, <user>), and one selector node (<name>). All instances of the <name> element in the same sibling set are selected in the filter output. The manager may need to know that <name> is used as an instance identifier in this particular data structure, but the server does not need to know that meta-data in order to process the request.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <top xmlns="http://example.com/schema/1.2/config">
        <users>
          <user>
            <name/>
          </user>
        </users>
      </top>
    </filter>
  </get-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <users>
        <user>
          <name>root</name>
        </user>
        <user>
          <name>fred</name>
        </user>
        <user>
          <name>barney</name>
        </user>
      </users>
    </top>
  </data>
</rpc-reply>
```

6.8.5 One specific <user> entry

This filter contains two containment nodes (<users>, <user>) and one content match node (<name>). All instances of the sibling set containing <name>, for which the value of <name> equals "fred", are selected in the filter output.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <top xmlns="http://example.com/schema/1.2/config">
        <users>
          <user>
            <name>fred</name>
          </user>
        </users>
      </top>
    </filter>
  </get-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <users>
        <user>
          <name>fred</name>
          <type>admin</type>
          <full-name>Fred Flintstone</full-name>
          <company-info>
            <dept>2</dept>
            <id>2</id>
          </company-info>
        </user>
      </users>
    </top>
  </data>
</rpc-reply>
```

6.8.6 Specific elements from a specific <user> entry

This filter contains two containment nodes (<users>, <user>), one content match node (<name>), and two selector nodes (<type>, <full-name>). All instances of the <type> and <full-name> elements in the same sibling set containing <name>, for which the value of <name> equals "fred", are selected in the filter output. The <company-info> element is not included because the sibling set contains selection nodes.

```
    <rpc message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <get-config>
        <source>
          <running/>
        </source>
        <filter type="subtree">
          <top xmlns="http://example.com/schema/1.2/config">
            <users>
              <user>
                <name>fred</name>
                <type/>
                <full-name/>
              </user>
            </users>
          </top>
        </filter>
      </get-config>
    </rpc>

    <rpc-reply message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <data>
        <top xmlns="http://example.com/schema/1.2/config">
          <users>
            <user>
              <name>fred</name>
              <type>admin</type>
              <full-name>Fred Flintstone</full-name>
            </user>
          </users>
        </top>
      </data>
    </rpc-reply>
```

6.8.7 Multiple Subtrees

This filter contains three subtrees (name=root, fred, barney)

The "root" subtree filter contains two containment nodes (<users>, <user>), one content match node (<name>), and one selector node (<company-info>). The subtree selection criteria is met, and just the company-info subtree for "root" is selected in the filter output.

The "fred" subtree filter contains three containment nodes (<users>, <user>, <company-info>), one content match node (<name>), and one

selector node (<id>). The subtree selection criteria is met, and just the <id> element within the company-info subtree for "fred" is selected in the filter output.

The "barney" subtree filter contains three containment nodes (<users>, <user>, <company-info>), two content match nodes (<name>, <type>), and one selector node (<dept>). The subtree selection criteria is not met because user "barney" is not a "superuser", and the entire subtree for "barney" (including its parent <user> entry) is excluded from the filter output.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-config>
      <source>
        <running/>
      </source>
      <filter type="subtree">
        <top xmlns="http://example.com/schema/1.2/config">
          <users>
            <user>
              <name>root</name>
              <company-info/>
            </user>
            <user>
              <name>fred</name>
              <company-info>
                <id/>
              </company-info>
            </user>
            <user>
              <name>barney</name>
              <type>superuser</type>
              <company-info>
                <dept/>
              </company-info>
            </user>
          </users>
        </top>
      </filter>
    </get-config>
  </rpc>

  <rpc-reply message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
      <top xmlns="http://example.com/schema/1.2/config">
        <users>
```

```
            <user>
              <name>root</name>
              <company-info>
                <dept>1</dept>
                <id>1</id>
              </company-info>
            </user>
            <user>
              <name>fred</name>
              <company-info>
                <id>2</id>
              </company-info>
            </user>
          </users>
        </top>
      </data>
    </rpc-reply>
```

6.8.8 Elements with attribute naming

In this example, the filter contains one containment node (<interfaces>), one attribute match expression (ifName), and one selector node (<interface>). All instances of the <interface> subtree which have an ifName attribute equal to "eth0" are selected in the filter output. The filter data elements and attributes must be qualified because the ifName attribute will not be considered part of the 'schema/1.2' namespace if it is unqualified.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get>
      <filter type="subtree">
        <t:interfaces xmlns:t="http://example.com/schema/1.2/stats">
          <t:interface t:ifName="eth0"/>
        </t:interfaces>
      </filter>
    </get>
  </rpc>

  <rpc-reply message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <data>
      <top xmlns="http://example.com/schema/1.2/config">
        <t:interfaces xmlns:t="http://example.com/schema/1.2/stats">
          <t:interface t:ifName="eth0">
            <t:ifInOctets>45621</t:ifInOctets>
            <t:ifOutOctets>774344</t:ifOutOctets>
          </t:interface>
        </t:interfaces>
      </top>
    </data>
  </rpc-reply>
```

If ifName were a child node instead of an attribute, then the following request would produce similar results.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get>
      <filter type="subtree">
        <interfaces xmlns="http://example.com/schema/1.2/stats">
          <interface>
            <ifName>eth0</ifName>
          </interface>
        </interfaces>
      </filter>
    </get>
  </rpc>
```

7. 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
- get-config
- edit-config
- copy-config
- delete-config
- lock
- unlock
- 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.

7.1 <get-config>

Description:

Retrieve all or part of a specified configuration.

Parameters:

source:

Name of the configuration datastore being queried, such as <running/>. If this element is unspecified, the <running/> configuration is used.

filter:

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

The filter element may optionally contain a "type" attribute. This attribute indicates the type of filtering syntax used within the filter element. The default filtering mechanism in NETCONF is referred to as subtree filtering and is described in Section 6. The value "subtree" explicitly identifies this type of filtering.

If the NETCONF peer supports the #xpath capability (Section 8.9), the value "xpath" may be used to indicate that the filter element contains an XPath expression.

Positive Response:

If the device can satisfy the request, the server sends an <rpc-reply> element containing a <data> 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: To retrieve the entire <users> subtree:

```
    <rpc message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <get-config>
        <source>
          <running/>
        </source>
        <filter type="subtree">
          <top xmlns="http://example.com/schema/1.2/config">
            <users/>
          </top>
        </filter>
      </get-config>
    </rpc>

    <rpc-reply message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <data>
        <top xmlns="http://example.com/schema/1.2/config">
          <users>
            <user>
              <name>root</name>
              <type>superuser</type>
              <full-name>Charlie Root</full-name>
              <company-info>
                <dept>1</dept>
                <id>1</id>
              </company-info>
            </user>
            <!-- additional <user> elements appear here... -->
          </users>
        </top>
      </data>
    </rpc-reply>
```

Section 6 contains additional examples of subtree filtering.

7.2 <edit-config>

Description:

The <edit-config> operation loads 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 will be created.

The device analyzes the source and target configurations and performs the requested changes. The target configuration is not necessarily replaced, as with the <copy-config> message. Instead the target configuration is changed in accordance with the source's data and requested operations.

Attributes:

operation:

Elements in the <config> subtree may contain an "operation" attribute. The attribute identifies the point in the configuration to perform the operation, and MAY appear on multiple elements throughout the <config> subtree.

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. This is the default behavior.

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/> or <candidate/>.

default-operation:

Selects the default operation (as described in the "operation" attribute) for this <edit-config> request. The default value for the default-operation parameter is "merge".

The default-operation parameter is optional, but if provided, must have one of the following values:

merge: The configuration data in the <config> parameter is merged with the configuration at the corresponding level in the target datastore. This is the default behavior.

replace: The configuration data in the <config> parameter completely replaces the configuration in the target datastore. This is useful for loading previously saved configuration data.

none: The target datastore is unaffected by the configuration in the <config> parameter, unless and until the incoming configuration data uses the "operation" attribute to request a different operation. If the configuration in the <config> parameter contains data for which there is not a corresponding level in the target datastore, an <rpc-error> is returned with an <error-tag> value of DATA_MISSING. Using "none" allows operations like "delete" to avoid unintentionally creating the parent hierarchy of the element to be deleted.

test-option:

The test-option element may be specified only if the device advertises the #validate capability (Section 8.6).

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 server 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 server to support the #rollback-on-error capability described in Section 8.5.

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="101"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
       <edit-config>
         <target>
           <running/>
         </target>
         <config>
           <top xmlns="http://example.com/schema/1.2/config">
             <interface>
               <name>Ethernet0/0</name>
               <mtu>1500</mtu>
             </interface>
           </top>
         </config>
       </edit-config>
     </rpc>

     <rpc-reply message-id="101"
          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="101"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
       <edit-config>
         <target>
           <running/>
         </target>
         <config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
           <top xmlns="http://example.com/schema/1.2/config">
             <interface xc:operation="replace">
               <name>Ethernet0/0</name>
               <mtu>1500</mtu>
               <address>
                 <name>1.2.3.4</name>
                 <prefix-length>24</prefix-length>
               </address>
             </interface>
           </top>
         </config>
       </edit-config>
     </rpc>

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

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

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

```
    <rpc message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <edit-config>
        <target>
          <running/>
        </target>
        <default-operation>none</default-operation>
        <config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
          <top xmlns="http://example.com/schema/1.2/config">
            <interface xc:operation="delete">
              <name>Ethernet0/0</name>
            </interface>
          </top>
        </config>
      </edit-config>
    </rpc>

    <rpc-reply message-id="101"
         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="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <edit-config>
        <target>
          <running/>
        </target>
        <default-operation>none</default-operation>
        <config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
          <top xmlns="http://example.com/schema/1.2/config">
            <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>
          </top>
        </config>
      </edit-config>
    </rpc>

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

7.3 <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, if allowed.

If a NETCONF peer supports the #url capability (Section 8.8), the <url> element can appear as the <source> or <target> parameter.

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, where both the <source> and <target> parameters use

the <url> element. If the source and target parameters identify the same URL or configuration datastore, an error MUST be returned with an error-tag containing INVALID_VALUE.

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="101"
     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="101"
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

7.4 <delete-config>

Description:

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

If a NETCONF peer supports the #url capability (Section 8.8), the <url> element can appear as the <target> parameter.

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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <delete-config>
    <target>
      <startup/>
    </target>
  </delete-config>
</rpc>

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

7.5 <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 CLI 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 failure of the underlying transport, or simple inactivity timeout. This criteria is dependent on the implementation and the underlying transport.

The lock operation takes a mandatory parameter, target. The target parameter names the configuration that 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 NETCONF 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 MUST 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 or rolled back

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.

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.

If the lock is already held, the <error-tag> element will be IN_USE and the <error-info> element will include the <session-id> of the lock owner. 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:

The following example shows a successful acquisition of a lock.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <lock>
    <target>
      <running/>
    </target>
  </lock>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/> <!-- lock succeeded -->
</rpc-reply>
```

Example:

The following example shows a failed attempt to acquire of a lock when the lock is already in use.

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <lock>
    <target>
      <running/>
    </target>
  </lock>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <rpc-error> <!-- lock failed -->
    <error-type>protocol</error-type>
    <error-tag>IN_USE</error-tag>
    <error-severity>error</error-severity>
    <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>
```

7.6 <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.

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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock>
    <target>
     <running/>
    </target>
  </unlock>
</rpc>

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

7.7 <get>

Description:

Retrieve running configuration and device state information.

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.

The filter element may optionally contain a "type" attribute. This attribute indicates the type of filtering syntax used within the filter element. The default filtering mechanism in NETCONF is referred to as subtree filtering and is described in Section 6. The value "subtree" explicitly identifies this type of filtering.

If the NETCONF peer supports the #xpath capability (Section 8.9), the value "xpath" may be used to indicate that the filter element contains an XPath expression.

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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <interfaces xmlns="http://example.com/schema/1.2/stats">
        <interface>
          <ifName>eth0</ifName>
        </interface>
      </interfaces>
    </filter>
  </get>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <interfaces xmlns="http://example.com/schema/1.2/stats">
        <interface>
          <ifName>eth0</ifName>
          <ifInOctets>45621</ifInOctets>
          <ifOutOctets>774344</ifOutOctets>
        </interface>
      </interfaces>
    </top>
  </data>
</rpc-reply>
```

7.8 <close-session>

Description:

Request graceful termination of a NETCONF session.

When a NETCONF server receives a <close-session> request, it will gracefully close the session. The server will release any locks and resources associated with the session and gracefully close any associated connections. Any NETCONF requests received after a <close-session> request will be ignored.

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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <close-session/>
</rpc>

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

7.9 <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, an 'INVALID_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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <kill-session>
      <session-id>4</session-id>
    </kill-session>
  </rpc>

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

8. 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 C. Future capability definitions may be published as standards by standards bodies or published as proprietary extensions.

A NETCONF capability is identified with a URI. The base capabilities are defined using URNs following the method described in RFC 3553 [8]. 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.

8.1 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 by each peer during

session establishment. When the NETCONF session is opened, each peer MUST send a <hello> element containing a list of that peer's capabilities. Each peer MUST send at least the base NETCONF capability, "urn:ietf:params:xml:ns:netconf:base:1.0". A server sending the <hello> element MUST include a <session-id> element containing the session ID for this NETCONF session.

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

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

8.2 Writable-Running Capability

8.2.1 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.

8.2.2 Dependencies

None.

8.2.3 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.

8.2.4 New Operations

None.

8.2.5 Modifications to Existing Operations

8.2.5.1 <edit-config>

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

8.2.5.2 <copy-config>

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

8.3 Candidate Configuration Capability

8.3.1 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 and 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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config> <!-- any NETCONF operation -->
    <source>
      <candidate/>
    </source>
  </get-config>
</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, e.g. #lock), 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.

8.3.2 Dependencies

None.

8.3.3 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.

8.3.4 New Operations

8.3.4.1 <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 device is unable to commit all of the changes in the candidate configuration datastore, then the running configuration MUST remain unchanged. If the device does succeed in committing, the running configuration MUST be updated with the contents of 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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>

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

8.3.4.2 <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 message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <discard-changes/>
</rpc>
```

This operation discards any uncommitted changes by reseting the candidate configuration with the content of the running configuration.

8.3.5 Modifications to Existing Operations

8.3.5.1 <lock> and <unlock>

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

```
  <rpc message-id="101"
       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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <unlock>
      <target>
        <candidate/>
      </target>
    </unlock>
  </rpc>
```

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.

8.4 Confirmed Commit Capability

8.4.1 Description

The #confirmed-commit capability indicates that the server will support the <confirmed> and <confirm-timeout> parameters for the <commit> protocol operation. See section Section 8.3 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.

8.4.2 Dependencies

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

8.4.3 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.

8.4.4 New Operations

None.

8.4.5 Modifications to Existing Operations

8.4.5.1 <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 follow-up commit (called the "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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <commit>
      <confirmed/>
      <confirm-timeout>20</confirm-timeout>
    </commit>
  </rpc>

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

8.5 Rollback on Error Capability

8.5.1 Description

This capability indicates that the server 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 used.

8.5.2 Dependencies

None

8.5.3 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.

8.5.4 New Operations

None.

8.5.5 Modifications to Existing Operations

8.5.5.1 <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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <error-option>rollback-on-error</error-option>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface>
          <name>Ethernet0/0</name>
          <mtu>100000</mtu>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

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

8.6 Validate Capability

8.6.1 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 test-option parameter to the <edit-config> operation and, when it is provided, checks at least for syntax errors.

8.6.2 Dependencies

None.

8.6.3 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.

8.6.4 New Operations

8.6.4.1 <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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <validate>
    <source>
      <candidate/>
    </source>
  </validate>
</rpc>

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

8.7 Distinct Startup Capability

8.7.1 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.

8.7.2 Dependencies

None.

8.7.3 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.

8.7.4 New Operations

None.

8.7.5 Modifications to Existing Operations

8.7.5.1 <copy-config>

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

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <copy-config>
      <source>
        <running/>
      </source>
      <target>
        <startup/>
      </target>
    </copy-config>
  </rpc>
```

8.8 URL Capability

8.8.1 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.

8.8.2 Dependencies

None.

8.8.3 Capability and Namespace

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

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

8.8.4 New Operations

None.

8.8.5 Modifications to Existing Operations

8.8.5.1 <edit-config>

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

8.8.5.2 <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.

8.8.5.3 <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.

8.8.5.4 <validate>

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

8.9 XPath Capability

8.9.1 Description

The XPath capability indicates that the NETCONF peer supports the use of XPath expressions in the <filter> element. XPath is described in [2].

8.9.2 Dependencies

None.

8.9.3 Capability and Namespace

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

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

The #xpath capability uses the base NETCONF namespace URN.

8.9.4 New Operations

None.

8.9.5 Modifications to Existing Operations

8.9.5.1 <get-config> and <get>

The #xpath capability modifies the <get> and <get-config> operations to accept the value "xpath" in the type attribute of the filter element. When the type attribute is set to "xpath", the contents of the filter element will be treated as an xpath expression and used to filter the returned data.

For example:

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <get-config>
      <source>
        <running/>
      </source>
      <filter type="xpath"> <!-- get the user named fred -->
        top/users/user[name="fred"]
      </filter>
    </get-config>
  </rpc>
```

9. Security Considerations

This document does not specify an authorization scheme, as such a scheme should be tied to a meta-data model or a data model. Implementators SHOULD provide a well thought out authorization scheme with NETCONF.

Authorization of individual users via the NETCONF server may or may not map 1:1 to other interfaces. First, the data models may be incompatible. Second, it may be desirable to authorize based on mechanisms available in the application protocol layer (TELNET, SSH, etc).

In addition, operations on configurations may have unintended consequences if those operations are also not guarded by the global lock on the files or objects being operated upon. For instance, a partially complete access list could be committed from a candidate

configuration unbeknownst to the owner of the lock of the candidate configuration, leading to either an insecure or inaccessible device if the lock on the candidate configuration does not also apply to the <copy-config> operation when applied to it.

Configuration information is by its very nature sensitive. Its transmission in the clear and without integrity checking leaves devices open to classic eavesdropping 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 operations 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. Such global operations MUST disallow the changing of information that an individual does not have authorization to perform. For example, if a user A is not allowed to configure an IP address on an interface but user B has configured an IP address on an interface in the <candidate> configuration, user A must not be allowed to commit the <candidate> configuration.

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 device's native user interface. These two mechanisms suffer from a race condition that may be ameliorated by removing the offending user from an AAA server. However, such a solution is not useful in all deployment scenarios, such as those where SSH public/private key pairs are used.

10. IANA Considerations

TBD.

11. 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

Phil Shafer, Juniper Networks

Steve Waldbusser

Margaret Wasserman, ThingMagic

The authors would like to acknowledge the members of the NETCONF working group. In particular, we would like to thank Wes Hardaker for his persistance and patience in assisting us with security considerations. We would also like to thank Randy Presuhn, Sharon Chisolm, Juergen Schoenwalder, Glenn Waters, David Perkins, Weijing Chen, Simon Leinen, Keith Allen and Dave Harrington for all of their valuable advice.

12. References

12.1 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] Clark, J. and S. DeRose, "XML Path Language (XPath) Version 1.0", W3C REC REC-xpath-19991116, November 1999.

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

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

[5] Berners-Lee, T., "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, June 1994.

[6] Moats, R., "URN Syntax", RFC 2141, May 1997.

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

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

12.2 Informative References

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

[10] 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 (editor)
Juniper Networks
1194 North Mathilda Ave
Sunnyvale, CA 94089
US

EMail: rpe@juniper.net

Appendix A. NETCONF Error List

```
Tag:         IN_USE
Error-type:  protocol, application
Severity:    error
Error-info:  none
Description: The request requires a resource that already in use.

Tag:         INVALID_VALUE
Error-type:  protocol, application
Severity:    error
Error-info:  none
Description: The request specifies an unacceptable value for one
             or more parameters.

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 server. 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">
  <!--
    <rpc> element
    -->
  <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"/>
```

```
    <!--
      data types and elements used to construct rpc-errors
      -->
    <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="INVALID_VALUE"/>
        <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:anyType" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <!--
      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"/>
    <!--
      <rpc-reply> element
      -->
    <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="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"/>
    <!--
      <test-option> parameter to <edit-config>
      -->
    <xs:simpleType name="test-optionType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="test-then-set"/>
        <xs:enumeration value="set"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="test-option" type="test-optionType"/>
    <!--
      <error-option> parameter to <edit-config>
      -->
    <xs:simpleType name="error-optionType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="stop-on-error"/>
        <xs:enumeration value="ignore-error"/>
        <xs:enumeration value="rollback-on-error"/>
```

```
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="error-option" type="error-optionType"/>
    <!--
      rpcOperationType: used as a base type for all
      NETCONF operations
      -->
    <xs:complexType name="rpcOperationType"/>
    <xs:element name="rpcOperation"
        type="rpcOperationType" abstract="true"/>
    <!--
      <config> element
      -->
    <xs:complexType name="config-inlineType">
      <xs:complexContent>
        <xs:extension base="xs:anyType"/>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="config" type="config-inlineType"/>
    <!--
      <data> element
      -->
    <xs:complexType name="data-inlineType">
      <xs:complexContent>
        <xs:extension base="xs:anyType"/>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="data" type="data-inlineType"/>
    <!--
      <filter> element
      -->
    <xs:simpleType name="FilterType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="subtree"/>
        <xs:enumeration value="xpath"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="filter-inlineType">
      <xs:complexContent>
        <xs:extension base="xs:anyType">
          <xs:attribute name="type"
            type="FilterType" default="subtree"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="filter" type="filter-inlineType"/>
    <!--
      configuration datastore names
```

```
      -->
    <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"/>
    <!--
      operation attribute used in <edit-config>
      -->
    <xs:simpleType name="EditOperationType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="merge"/>
        <xs:enumeration value="replace"/>
        <xs:enumeration value="create"/>
        <xs:enumeration value="delete"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:attribute name="operation"
      type="EditOperationType" default="merge"/>
    <!--
      <default-operation> element
      -->
    <xs:simpleType name="DefaultOperationType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="merge"/>
        <xs:enumeration value="replace"/>
        <xs:enumeration value="none"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="default-operation" type="DefaultOperationType"/>
    <!--
      <url> element
      -->
    <xs:complexType name="config-uriType">
      <xs:simpleContent>
        <xs:extension base="xs:anyURI"/>
      </xs:simpleContent>
    </xs:complexType>
    <xs:element name="url" type="config-uriType"/>
    <!--
      <source> element
      -->
    <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"/>
    <!--
      <target> element
      -->
    <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"/>
    <!--
      <get-config> operation
      -->
    <xs:complexType name="get-configType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="source"/>
            <xs:element ref="filter" minOccurs="0" maxOccurs="1"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="get-config" type="get-configType"
      substitutionGroup="rpcOperation"/>
    <!--
      <edit-config> operation
      -->
    <xs:complexType name="edit-configType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="target"/>
            <xs:element ref="default-operation" minOccurs="0"/>
            <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"/>
    <!--
      <copy-config> operation
      -->
    <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"/>
    <!--
      <delete-config> operation
      -->
    <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"/>
    <!--
      <get> operation
      -->
    <xs:complexType name="getType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="filter" minOccurs="0" maxOccurs="1"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="get" type="getType"
      substitutionGroup="rpcOperation"/>
    <!--
      <lock> operation
      -->
    <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"/>
    <!--
      <unlock> operation
      -->
    <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"/>
    <!--
      <validate> operation
      -->
    <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"/>
    <!--
      <commit> operation
      -->
    <xs:complexType name="commitType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element name="confirmed" minOccurs="0"/>
            <xs:element name="confirm-timeout" minOccurs="0"/>
          </xs:sequence>
```

```
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="commit" type="commitType"
      substitutionGroup="rpcOperation"/>
    <!--
      <discard-changes> operation
      -->
    <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"/>
    <!--
      <close-session> operation
      -->
    <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"/>
    <!--
      <kill-session> operation
      -->
    <xs:complexType name="kill-sessionType">
      <xs:complexContent>
        <xs:extension base="rpcOperationType">
          <xs:sequence>
            <xs:element ref="session-id" minOccurs="0" maxOccurs="1" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:element name="kill-session" type="kill-sessionType"
      substitutionGroup="rpcOperation"/>
    <!--
      <hello> element
      -->
    <xs:element name="capability" type="xs:anyURI" />
    <xs:element name="capabilities">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="capability" maxOccurs="unbounded" />
        </xs:sequence>
```

```
      </xs:complexType>
    </xs:element>
    <xs:element name="hello">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="capabilities" maxOccurs="1" />
          <xs:element ref="session-id" minOccurs="0" maxOccurs="1" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

Appendix C. Capability Template

C.1 capability-name (template)

C.1.1 Overview

C.1.2 Dependencies

C.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.

C.1.4 New Operations

C.1.4.1 <op-name>

C.1.5 Modifications to Existing Operations

C.1.5.1 <op-name>

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

C.1.6 Interactions with Other Capabilities

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

Appendix D. Configuring Multiple Devices with NETCONF

D.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.

D.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.

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

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <lock>
      <target>
        <running/>
      </target>
```

```
      </lock>
    </rpc>
```

D.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="101"
       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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <edit-config>
      <target>
        <candidate/>
      </target>
      <config>
        <!-- place incoming configuration here -->
      </config>
    </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.

D.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="101"
       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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <validate>
      <source>
        <candidate/>
      </source>
    </validate>
  </rpc>
```

D.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="101"
       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.

D.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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <edit-config>
      <target>
        <running/>
      </target>
      <config>
        <url>file://incoming.conf</url>
      </config>
    </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="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <commit>
      <confirmed/>
      <confirm-timeout>15</confirm-timeout>
    </commit>
  </rpc>
```

D.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).

D.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="101"
     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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>
```

D.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="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <unlock>
    <target>
      <running/>
    </target>
  </unlock>
</rpc>
```

D.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 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 E. Deferred Features

The following features have been deferred until a future revision of this document.

- Granular locking of configuration objects.
- Named configuration files/datastores.
- Support for multiple NETCONF channels.
- Asynchronous notifications.

Appendix F. Change Log

RFC Editor: Please remove this section before RFC publication.

F.1 draft-ietf-netconf-prot-05

- Change XPATH to XPath.
- Fix I-D nits (mostly long lines).
- Remove "--" from XSD comments.
- Add <source> attribute where it was missing in <get-config> examples.
- Clarified Section 8.1 by indicating that each peer MUST send a <hello> element at session startup.
- Typo propriety -> proprietary in Section 8.
- Fix some bugs in examples.
- Section 7.1: typo: change <config> to <data> in the positive response section.
- Section 7.1: If <filter> is unspecified, the entire configuration is returned. If it is empty, nothing is returned.
- Be explicit about <commit> being atomic.
- s/MAY/SHOULD/ wrt supporting more than one NETCONF session.

- Strengthen language to say that NETCONF requests MUST be processed serially.
- Fix misspelling of "unbeknownst."
- Change "Expect scripts" to "CLI scripts" in Section 7.5.
- Change "system software" to "device" in Section 1.3.
- The <hello> element must also include the session ID (issue I002).
- Address all accepted clarifications from working group last call. See the NETCONF mailing list for details.
- Address all closed issues from working group last call. See the NETCONF mailing list for details.

F.2 draft-ietf-netconf-prot-04

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/.

- Update security considerations (action from IETF 60).
- Add type attribute on filter element (issue 14.1).
- Add #xpath capability (issue 14.1).
- <rpc-reply> for <get-config> returns <data> element, not <config> element (issue 14.1).
- Add detailed description of subtree filtering (issue 14.1.2).
- Typo: change confirmed-timeout -> confirm-timeout in XSD.
- Typo: correct misnaming of test-option parameter in text for the validate capability.
- <target> is now a mandatory parameter for <lock> and <unlock>. There is no default target (action from IETF 60).
- Remove XML schema for NETCONF state data (action from IETF 60).
- Correct namespace handling a number of examples. The fix is to put the device's configuration under a top level tag called <top> which is in the device's namespace.

- Use message-id 101 everywhere.
- Add default-operation parameter to <edit-config> (action from IETF 60).
- Fix <edit-config> examples in Appendix D.
- Update and reformat protocol XSD.
- Remove XML usage guidelines. Add a section on XML considerations covering the NETCONF namespace and no DTD restriction (action from IETF 60).

F.3 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.4 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 D. 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

Disclaimer of Validity

Copyright Statement

Acknowledgment

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