Network Working Group
Internet-Draft
Updates: 6020 (if approved)
Intended status: Standards Track
Expires: April 8, 2013

M. Bjorklund
Tail-f Systems
L. Lhotka
CZ.NIC

October 5, 2012

Operational Data in NETCONF and YANG
draft-bjorklund-netmod-operational-00

Abstract

This document defines the concept of operational state data in the context of YANG and the Network Configuration Protocol (NETCONF). It updates RFC 6020 with rules for how to model the operational state, and defines NETCONF operations to retrieve and modify the operational state.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 3
1.1. Terminology 3
1.1.1. Terms 3
2. Objectives 4
3. Problem Statement 5
3.1. Modeling and Retrieving Operational State 5
3.1.1. Example: Interface List 5
3.2. Modifying the Operational State 6
3.2.1. Example: Routing Table Modification 7
4. Datastores 8
4.1. Operational State Datastore 8
4.2. Configuration Datastore 9
5. Constraints 10
5.1. Alternative A 10
5.2. Alternative B 10
5.3. Alternative C 10
6. Protocol Operations 11
6.1. <get-operational> 11
6.1.1. Example: Ethernet Duplex 11
6.2. <edit-operational> 12
7. YANG Module 13
8. IANA Considerations 15
9. Security Considerations 16
10. References 17
10.1. Normative References 17
10.2. Informative References 17
Appendix A. Example: Interface List 18
Appendix B. Example: Ethernet Duplex 19
Appendix C. Example: Admin vs. Oper State 20
Authors' Addresses 21

1. Introduction

1.1. Terminology

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, [RFC2119].

1.1.1. Terms

The following terms are defined in [RFC6241] and are not redefined here:

o client

o configuration datastore

o datastore

o server

The following terms are defined in [RFC6020] and are not redefined here:

o data model

o schema tree

o data node

The following terms are used within this document:

o operational state data: The data in the operational state datastore.

o operational state datastore: A conceptual data structure from which one can determine device state and behavior.

2. Objectives

- Develop a general model applicable not only to NETCONF but also to other approaches (RESTful, editable state data etc.).

- Develop a specific model for NETCONF and YANG.

- As little changes to NETCONF and YANG as possible.

- Clarification of the terms "operational state data" and "configuration".

3. Problem Statement

3.1. Modeling and Retrieving Operational State

The NETCONF operation <get> returns both device state data and the running configuration. Quite often, device parameters require a dual representation, both as configuration and state data.

For instance, an IP address may be specified in an interface configuration but, depending on other circumstances, this address may not be used for that interface. In any case, an operator should be able to obtain the addresses that are in operational use.

This implies that some state data must be modeled separately from the configuration data, which leads to a certain amount of duplication in data models. This approach has other drawbacks, too. It is counter-intuitive to data model designers, for whom configuration and state parameters are closely related (see Section 3.1.1 for an example). Data model duplication is error prone and leads to bigger data models, that are more difficult to understand. Further, there is no formal information in the data model about the relationship between the configuration and operational state data.

3.1.1. Example: Interface List

Suppose we want to model a list of interfaces. We allow pre-configuration, i.e., it is legal to configure an interface for which there is currently no hardware present in the system. In this simple example, each interface has a name and a counter of the number of packets received. The counter is operational state data.

```
list interface {
  key name;

  leaf name { ... }
  leaf in-packets {
    type yang:counter64;
    config false;
  }
  ...
}
```

A particular device has hardware for two interfaces with names "eth0" and "eth1". In the configuration there is:

```
      <interface>
        <name>eth0</name>
        ...
      </interface>
      <interface>
        <name>eth2</name>
        ...
      </interface>
```

We can see this by doing <get-config>.

Operationally, however, the interfaces used are "eth0" and "eth1", although "eth1" does not have any configuration and does not send or receive packets.

How can an operator learn about the presence of "eth1"? The <get> operation returns the running configuration and state data together. So, <get> will not show "eth1", since it is not present in the running configuration.

With NETCONF as currently defined, the only alternative is to duplicate the data model:

```
       list interface {
         key name;

         leaf name { ...}
         ...
       }

       list interface-oper {
         config false;
         key name;

         leaf name { ...}
         leaf in-packets { ... }
         ...
       }
```

3.2. Modifying the Operational State

In some cases, it is useful for clients to directly modify the operational state. An example of this is the recent discussions around an Interface to the Routing System (IRS), where a client needs to modify the routing table, without storing routes in the configuration.

With NETCONF as currently designed, the only way to do this is to

define separate rpc operations. This leads to another kind of data model duplication, where every writable parameter is modeled both as state data that can be retrieved using the <get> operation, and also as input parameters to at least one rpc operation.

3.2.1. Example: Routing Table Modification

Suppose we want to model IPv4 routing tables as operational state, and we also want to be able to let a client modify this data. We have to do:

```
list routing-table {
  config false;
  key name;

  leaf name { ... }
  list route {
    key id;

    leaf id { ... }
    leaf dest-prefix { ... }
    leaf next-hop { ... }
    ...
  }
}

rpc add-route {
  input {
    leaf routing-table-name { ... }
    leaf route-id { ... }
    leaf dest-prefix { ... }
    leaf next-hop { ... }
  }
}

rpc delete-route {
  input {
    leaf routing-table-name { ... }
    leaf route-id { ... }
  }
}
```

4. Datastores

The fundamental idea of this document is to define operational state data as an explicit data structure called the operational state datastore. It is available to all management interfaces, which includes NETCONF but also other interfaces such as SNMP.

The "running" configuration datastore is viewed as a separate overlay data structure whose layout is identical to the subset of the operational state datastore that represents configuration.

4.1. Operational State Datastore

The operational state datastore consists of all parameters that provide information about the instantaneous state of the device and immediately influence the device's behavior.

The operational state datastore is a conceptual data structure. This means that implementations may choose any suitable representation of the datastore, or even generate it dynamically upon request.

Operational state may be modified through one or more management interfaces, or through the operation of network protocols. All such means of accessing and changing the operational state act conceptually on the same data - the operational state datastore. It means, for instance, that any change caused by a network protocol is immediately visible to all management interfaces.

The schema for the operational state datastore is made up of all data nodes defined in YANG modules, specifically both "config true" and "config false" data nodes.

Note that when <get-config> is used to retrieve a "config true" node, the value stored in the configuration datastore is returned. When <get-operational> is used to retrieve the same node, the value actually used by the device is returned. This value may or may not be the same as the value in the datastore.

```
+------------------------------------------------------------------+
| Open Question                                                    |
+------------------------------------------------------------------+
| Should there be a YANG statement 'operational <bool>' so that     |
| config true nodes can be marked as not being part of the          |
| operational schema?                                              |
+------------------------------------------------------------------+
```

Nodes in the operational data store cannot be directly modified using the standard NETCONF operations.

```
+--------------------------------------------------------------------+
| Open Question                                                      |
+--------------------------------------------------------------------+
| introduce oper:writable for nodes that can be modified by          |
| <edit-operational>.                                                |
+--------------------------------------------------------------------+
```

4.2. Configuration Datastore

The configuration datastore can be thought of as a blueprint for the operational datastore. Specifically, the schema for the configuration datastore is congruent to a subtree of the schema for operational state datastore containing only "config true" nodes. In other words, every data node in the configuration datastore schema has a corresponding data node with the same name in the operational datastore schema, and the latter data node is "config true".

Whenever a NETCONF client modifies the configuration datastore, the server MUST immediately attempt to project the changes into the operational state datastore. Most of the time, it simply means copying the values in the configuration datastore to the corresponding nodes in the operational state datastore. If a leaf's default value is in use (See section 7.6.1. of RFC 6020), the default value is copied to the operational data store.

The configuration datastore may contain data nodes that are not projected into the operational state datastore. This happens in the following three scenarios:

1. Pre-provisioned configuration prepared for hardware components that are not yet present in the device. See Section 3.1.1 above.

2. Pre-provisioned configuration for components (hardware, protocols, etc.) that are intended to replace an existing item but are of a different type.

3. Parts of configuration that are momentarily not applicable.

5. Constraints

This document updates section 8 of RFC 6020 with rules for the operational state datastore.

NOTE: The rest of this section documents some alternatives that the authors want to discuss

There are a couple of design alternatives here:

5.1. Alternative A

No constraints ("must", "mandatory", "unique", "min-elements", "max-elements") are enforced on the operational state datastore. For example, this means that a mandatory "config true" leaf does not have to be present in the operational state datastore.

The problem with this approach is that there is no way to formally define constraints on the OSD in the data model. This may be needed in order to allow for coexistence of NETCONF with other management interfaces that do not use the configuration datastore. Such constraints can be specified in description statement though.

5.2. Alternative B

Change the definitions of mandatory, must, to work on osd instead of config.

This would be a major backwards incompatible change to YANG, and it would not be possible to define constraints on the configuration.

5.3. Alternative C

Introduce new YANG statements for OSD constraints, e.g. osd:must, osd:mandatory etc.

The drawback with this is that it adds complexity.

6. Protocol Operations

6.1. <get-operational>

This document introduces a new operation <get-operational>, which is used to retrieve the operational state data from a device. Note how this operation differs from <get>, which is used to retrieve both the running configuration and state data.

<get-operational> takes the same parameters as <get>.

Since leafs with default values defined in the data model are always explicitly set in the operational data store, there is no need for :with-defaults handling in the <get-operational> operation.

6.1.1. Example: Ethernet Duplex

As an example, consider a very simplified data model with a single leaf for ethernet duplex:

```
leaf duplex {
  type enumeration {
    enum "half";
    enum "full";
    enum "auto";
  }
  config true;
}
```

Suppose a device with this data model implements the candidate datastore. The following is an example of data from such a device:

```
get-config from candidate:


  <duplex>half</duplex>


get-config from running:


  <duplex>auto</duplex>


get-operational:


  <duplex>full</duplex>
```

In this example, the running configuration tells the device to negotiate the duplex mode, and the current, operationally used, value is "full". At the same time, the (uncommitted) candidate configuration contains the value "half".

6.2. <edit-operational>

[Editor's note: NOT FINISHED - not clear if we need this]

Introduce edit-operational. This modifies the subset of the operational data tree that is also marked as writable.

Drawback: does not handle persistent operational data. If we have persistent operational data, this has to be its own data store that can be read and written.

A data model w/o the writable markers cannot be written to. This is a problem, since it is not obvious that the original designer though about future use cases. For example, our route tables are read-only. Then IRS comes along and wants to write to this data. Do we have to update our spec? Not good. One option is for IRS to publish a deviation data model that added the writable statement to our model. This would be backwards compliant and good. Even better would be if they could publish a conformance statement in a module, w/o the need for deviations.

7. YANG Module

RFC Ed.: update the date below with the date of RFC publication and remove this note.

```
<CODE BEGINS> file "ietf-netconf-operational.yang"

module ietf-netconf-operational {

  namespace "urn:ietf:params:xml:ns:yang:ietf-netconf-operational";
  prefix "oper";

  import ietf-yang-types {
    prefix yang;
  }
  import ietf-inet-types {
    prefix inet;
  }
  import ietf-netconf {
    prefix nc;
  }

  rpc get-operational {
    input {
      choice filter-spec {
        anyxml subtree-filter {
          description
            "This parameter identifies the portions of the
             operational state datastore to retrieve.";
          reference "RFC 6241, Section 6.";
        }
        leaf xpath-filter {
          if-feature nc:xpath;
          type yang:xpath1.0;
          description
            "This parameter contains an XPath expression
             identifying the portions of the operational state
             datastore to retrieve.";
        }
      }
    }

    output {
      anyxml data {
        description
          "Copy of the operational state data that matched the filter
           criteria (if any).  An empty data container indicates that
           the request did not produce any results.";
```

```
           }
         }
       }

       rpc edit-operational {
         input {
           leaf default-operation {
             type enumeration {
               enum merge {
                 description
                   "The default operation is merge.";
               }
               enum replace {
                 description
                   "The default operation is replace.";
               }
               enum none {
                 description
                   "There is no default operation.";
               }
             }
             default "merge";
             description
               "The default operation to use.";
           }

           choice edit-content {
             mandatory true;
             description
               "The content for the edit operation.";

             anyxml data {
               description
                 "Inline data content.";
             }
             leaf url {
               if-feature nc:url;
               type inet:uri;
               description
                 "URL-based config content.";
             }
           }
         }
       }
     }

     <CODE ENDS>
```

8. IANA Considerations

This document registers a URI in the IETF XML registry [RFC3688]. Following the format in RFC 3688, the following registration is requested to be made.

URI: urn:ietf:params:xml:ns:yang:ietf-netconf-operational

Registrant Contact: The NETMOD WG of the IETF.

XML: N/A, the requested URI is an XML namespace.

This document registers a YANG module in the YANG Module Names registry [RFC6020].

```
  name:         ietf-netconf-operational
  namespace:    urn:ietf:params:xml:ns:yang:ietf-netconf-operational
  prefix:       oper
  reference:    RFC XXXX
```

9. Security Considerations

This document does not introduce any new security concerns in addition to those specified in [RFC6020] and [RFC6241].

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC6020] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011.

10.2. Informative References

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

Appendix A. Example: Interface List

With the proposed solution, the interface list example from ^ex-if-list-2, can be solved with a single list:

```
  list interface {
    key name;

    leaf name { ... }
    leaf in-packets {
      type yang:counter64;
      config false;
    }
    ...
  }
```

The operation <get-operational> will return the interfaces available on the device:

```
   <interface>
     <name>eth0</name>
     ...
   </interface>
   <interface>
     <name>eth1</name>
     ...
   </interface>
```

And <get-config> on running will return the configured interfaces, just as before:

```
   <interface>
     <name>eth0</name>
     ...
   </interface>
   <interface>
     <name>eth2</name>
     ...
   </interface>
```

Appendix B. Example: Ethernet Duplex

A typical problem is when the value space for the configuration data is a super set of the value space for the operational state data. An example of this is Ethernet duplex, which can be configured as "half", "full", or "auto", but the operationally used value is either "half" or "full". Without the definition of operational state in this document, this would have to be modeled as two separate leafs:

```
  leaf duplex {
    type enumeration {
      enum "half";
      enum "full";
      enum "auto";
    }
  }

  leaf oper-duplex {
    type enumeration {
      enum "half";
      enum "full";
    }
  }
```

With the solution defined in this document, a single leaf is sufficient:

```
  leaf duplex {
    type enumeration {
      enum "half";
      enum "full";
      enum "auto";
    }
  }
```

Appendix C. Example: Admin vs. Oper State

Another common problem is when the value space for the configured data is a subset of the operational state data. An example is an interface's desired state, and its operational state. The desired state can be "up" or "down", but the operational state can be "up", "lower-layer-down", "testing", etc.

These kind of situations are still best modeled as two separate leafs, one "admin-state" and one "oper-state".

Authors' Addresses

Martin Bjorklund
Tail-f Systems

Email: mbj@tail-f.com

Ladislav Lhotka
CZ.NIC

Email: lhotka@nic.cz