Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 4, 2018

M. Bjorklund
Tail-f Systems
J. Schoenwaelder
Jacobs University
P. Shafer
K. Watsen
Juniper Networks
R. Wilton
Cisco Systems
July 3, 2017

Network Management Datastore Architecture
draft-ietf-netmod-revised-datastores-03

Abstract

Datastores are a fundamental concept binding the data models written in the YANG data modeling language to network management protocols such as NETCONF and RESTCONF. This document defines an architectural framework for datastores based on the experience gained with the initial simpler model, addressing requirements that were not well supported in the initial model.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document provides an architectural framework for datastores as they are used by network management protocols such as NETCONF [RFC6241], RESTCONF [RFC8040] and the YANG [RFC7950] data modeling language. Datastores are a fundamental concept binding network management data models to network management protocols. Agreement on a common architectural model of datastores ensures that data models can be written in a network management protocol agnostic way. This architectural framework identifies a set of conceptual datastores but it does not mandate that all network management protocols expose all these conceptual datastores. This architecture is agnostic with regard to the encoding used by network management protocols.

2. Terminology

This document defines the following terms:

o datastore: A conceptual place to store and access information. A datastore might be implemented, for example, using files, a database, flash memory locations, or combinations thereof. A datastore maps to an instantiated YANG data tree.

o configuration: Data that is required to get a device from its initial default state into a desired operational state. This data is modeled in YANG using "config true" nodes. Configuration can originate from different sources.

o configuration datastore: A datastore holding configuration.

o running configuration datastore: A configuration datastore holding the current configuration of the device. It may include inactive configuration or template-mechanism-oriented configuration that require further expansion. This datastore is commonly referred to as "<running>".

o candidate configuration datastore: A configuration datastore that can be manipulated without impacting the device's running configuration datastore and that can be committed to the running configuration datastore. This datastore is commonly referred to as "<candidate>".

- startup configuration datastore: A configuration datastore holding the configuration loaded by the device into the running configuration datastore when it boots. This datastore is commonly referred to as "<startup>".

- intended configuration: Configuration that is intended to be used by the device. For example, intended configuration excludes any inactive configuration and it would include configuration produced through the expansion of templates.

- intended configuration datastore: A configuration datastore holding the complete intended configuration of the device. This datastore is commonly referred to as "<intended>".

- conventional configuration datastore: One of the following set of configuration datastores: <running>, <startup>, <candidate>, and <intended>. These datastores share a common schema and protocol operations allow copying data between these datastores. The term "conventional" is chosen as a generic umbrella term for these datastores.

- conventional configuration: Configuration that is stored in any of the conventional configuration datastores.

- dynamic datastore: A datastore holding data obtained dynamically during the operation of a device through interaction with other systems, rather than through one of the conventional configuration datastores.

- dynamic configuration: Configuration obtained via a dynamic datastore.

- learned configuration: Configuration that has been learned via protocol interactions with other systems that is not conventional or dynamic configuration.

- system configuration: Configuration that is supplied by the device itself.

- default configuration: Configuration that is not explicitly provided but for which a value defined in the data model is used.

- applied configuration: Configuration that is actively in use by a device. Applied configuration originates from conventional, dynamic, learned, system and default configuration.

- system state: The additional data on a system that is not configuration, such as read-only status information and collected

statistics. System state is transient and modified by interactions with internal components or other systems. System state is modeled in YANG using "config false" nodes.

- o operational state: The combination of applied configuration and system state.

- o operational state datastore: A datastore holding the complete operational state of the device. This datastore is commonly referred to as "<operational>".

- o origin: A metadata annotation indicating the origin of a data item.

- o remnant configuration: Configuration that remains part of the applied configuration for a period of time after it has been removed from the intended configuration or dynamic configuration. The time period may be minimal, or may last until all resources used by the newly-deleted configuration (e.g., network connections, memory allocations, file handles) have been deallocated.

The following additional terms are not datastore specific but commonly used and thus defined here as well:

- o client: An entity that can access YANG-defined data on a server, over some network management protocol.

- o server: An entity that provides access to YANG-defined data to a client, over some network management protocol.

- o notification: A server-initiated message indicating that a certain event has been recognized by the server.

- o remote procedure call: An operation that can be invoked by a client on a server.

3. Background

NETCONF [RFC6241] provides the following definitions:

- o datastore: A conceptual place to store and access information. A datastore might be implemented, for example, using files, a database, flash memory locations, or combinations thereof.

- o configuration datastore: The datastore holding the complete set of configuration that is required to get a device from its initial default state into a desired operational state.

YANG 1.1 [RFC7950] provides the following refinements when NETCONF is used with YANG (which is the usual case but note that NETCONF was defined before YANG existed):

- o datastore: When modeled with YANG, a datastore is realized as an instantiated data tree.

- o configuration datastore: When modeled with YANG, a configuration datastore is realized as an instantiated data tree with configuration.

[RFC6244] defined operational state data as follows:

- o Operational state data is a set of data that has been obtained by the system at runtime and influences the system's behavior similar to configuration data. In contrast to configuration data, operational state is transient and modified by interactions with internal components or other systems via specialized protocols.

Section 4.3.3 of [RFC6244] discusses operational state and among other things mentions the option to consider operational state as being stored in another datastore. Section 4.4 of this document then concludes that at the time of the writing, modeling state as distinct leafs and distinct branches is the recommended approach.

Implementation experience and requests from operators [I-D.ietf-netmod-opstate-reqs], [I-D.openconfig-netmod-opstate] indicate that the datastore model initially designed for NETCONF and refined by YANG needs to be extended. In particular, the notion of intended configuration and applied configuration has developed.

Furthermore, separating operational state from configuration in a separate branch in the data model has been found operationally complicated, and typically impacts the readability of module definitions due to overuse of groupings. The relationship between the branches is not machine readable and filter expressions operating on configuration and on related operational state are different.

3.1. Original Model of Datastores

The following drawing shows the original model of datastores as it is currently used by NETCONF [RFC6241]:

```
     +-------------+                 +-----------+
     | <candidate> |                 | <startup> |
     |  (ct, rw)   |<---+       +--->| (ct, rw)  |
     +-------------+    |       |    +-----------+
            |           |       |           |
            |         +-----------+         |
            +-------->| <running> |<--------+
                      | (ct, rw)  |
                      +-----------+
                            |
                            v
                     operational state  <--- control plane
                         (cf, ro)

     ct = config true; cf = config false
     rw = read-write; ro = read-only
     boxes denote datastores
```

Note that this diagram simplifies the model: read-only (ro) and read-write (rw) is to be understood at a conceptual level. In NETCONF, for example, support for <candidate> and <startup> is optional and <running> does not have to be writable. Furthermore, <startup> can only be modified by copying <running> to <startup> in the standardized NETCONF datastore editing model. The RESTCONF protocol does not expose these differences and instead provides only a writable unified datastore, which hides whether edits are done through <candidate> or by directly modifying <running> or via some other implementation specific mechanism. RESTCONF also hides how configuration is made persistent. Note that implementations may also have additional datastores that can propagate changes to <running>. NETCONF explicitly mentions so called named datastores.

Some observations:

- Operational state has not been defined as a datastore although there were proposals in the past to introduce an operational state datastore.

- The NETCONF <get/> operation returns the content of the <running> configuration datastore together with the operational state. It is therefore necessary that "config false" data is in a different branch than the "config true" data if the operational state can have a different lifetime compared to configuration or if configuration is not immediately or successfully applied.

- Several implementations have proprietary mechanisms that allow clients to store inactive data in <running>; this inactive data is only exposed to clients that indicate that they support the

concept of inactive data; clients not indicating support for inactive data receive the content of <running> with the inactive data removed. Inactive data is conceptually removed before validation.

- Some implementations have proprietary mechanisms that allow clients to define configuration templates in <running>. These templates are expanded automatically by the system, and the resulting configuration is applied internally.

- Some operators have reported that it is essential for them to be able to retrieve the configuration that has actually been successfully applied, which may be a subset or a superset of the <running> configuration.

4. Architectural Model of Datastores

Below is a new conceptual model of datastores extending the original model in order to reflect the experience gained with the original model.

```
     +-------------+                 +-----------+
     | <candidate> |                 | <startup> |
     |  (ct, rw)   |<---+       +--->| (ct, rw)  |
     +-------------+    |       |    +-----------+
            |           |       |           |
            |         +-----------+         |
            +-------->| <running> |<--------+
                      | (ct, rw)  |
                      +-----------+
                            |
                            |        // configuration transformations,
                            |        // e.g., removal of "inactive"
                            |        // nodes, expansion of templates
                            v
                      +------------+
                      | <intended> | // subject to validation
                      | (ct, ro)   |
                      +------------+
                            |        // changes applied, subject to
                            |        // local factors, e.g., missing
                            |        // resources, delays
                            |
                            |   +-------- learned configuration
       dynamic              |   +-------- system configuration
       datastores -----+    |   +-------- default configuration
                       |    |   |
                       v    v   v
                    +---------------+
                    | <operational> | <-- system state
                    | (ct + cf, ro) |
                    +---------------+

     ct = config true; cf = config false
     rw = read-write; ro = read-only
     boxes denote named datastores
```

4.1. The Startup Configuration Datastore (<startup>)

The startup configuration datastore (<startup>) is an optional configuration datastore holding the configuration loaded by the device when it boots. <startup> is only present on devices that separate the startup configuration from the running configuration datastore.

The startup configuration datastore may not be supported by all protocols or implementations.

On devices that support non-volatile storage, the contents of <startup> will typically persist across reboots via that storage. At boot time, the device loads the saved startup configuration into <running>. To save a new startup configuration, data is copied to <startup>, either via implicit or explicit protocol operations.

4.2. The Candidate Configuration Datastore (<candidate>)

The candidate configuration datastore (<candidate>) is an optional configuration datastore that can be manipulated without impacting the device's current configuration and that can be committed to <running>.

The candidate configuration datastore may not be supported by all protocols or implementations.

<candidate> does not typically persist across reboots, even in the presence of non-volatile storage. If <candidate> is stored using non-volatile storage, it should be reset at boot time to the contents of <running>.

4.3. The Running Configuration Datastore (<running>)

The running configuration datastore (<running>) holds the complete current configuration on the device. It may include inactive configuration or template-mechanism-oriented configuration that require further expansion.

If a device does not have a distinct <startup> and non-volatile is available, the device will typically use that non-volatile storage to allow <running> to persist across reboots.

4.4. The Intended Configuration Datastore (<intended>)

The intended configuration datastore (<intended>) is a read-only configuration datastore. It is tightly coupled to <running>. When data is written to <running>, the data that is to be validated is also conceptually written to <intended>. Validation is performed on the contents of <intended>.

For simple implementations, <running> and <intended> are identical.

<intended> does not persist across reboots; its relationship with <running> makes that unnecessary.

Currently there are no standard mechanisms defined that affect <intended> so that it would have different contents than <running>, but this architecture allows for such mechanisms to be defined.

One example of such a mechanism is support for marking nodes as inactive in <running>. Inactive nodes are not copied to <intended>, and are thus not taken into account when validating the configuration.

Another example is support for templates. Templates are expanded when copied into <intended>, and the expanded result is validated.

4.5. Conventional Configuration Datastores

The conventional configuration datastores are a set of configuration datastores that share a common schema, allowing data to be copied between them. The term is meant as a generic umbrella description of these datastores. The set of datastores include:

o <running>

o <candidate>

o <startup>

o <intended>

Other conventional configuration datastores may be defined in future documents.

The flow of data between these datastores is depicted in section Section 4.

The specific protocols may define explicit operations to copy between these datastores, e.g., NETCONF's <copy-config> operation.

4.6. Dynamic Datastores

The model recognizes the need for dynamic datastores that are, by definition, not part of the persistent configuration of a device. In some contexts, these have been termed ephemeral datastores since the information is ephemeral, i.e., lost upon reboot. The dynamic datastores interact with the rest of the system through <operational>.

4.7. The Operational State Datastore (<operational>)

The operational state datastore (<operational>) is a read-only datastore that consists of all "config true" and "config false" nodes defined in the schema. In the original NETCONF model the operational state only had "config false" nodes. The reason for incorporating

"config true" nodes here is to be able to expose all operational settings without having to replicate definitions in the data models.

<operational> contains system state and all configuration actually used by the system. This includes all applied configuration from <intended>, system-provided configuration, and default values defined by any supported data models. In addition, <operational> also contains applied data from dynamic datastores.

Requests to retrieve nodes from <operational> always return the value in use if the node exists, regardless of any default value specified in the YANG module. If no value is returned for a given node, then this implies that the node is not used by the device.

<operational> does not persist across reboots.

Changes to configuration may take time to percolate through to <operational>. During this period, <operational> may contain nodes for both the previous and current configuration, as closely as possible tracking the current operation of the device. Such remnant configuration from the previous configuration persists until the system has released resources used by the newly-deleted configuration (e.g., network connections, memory allocations, file handles).

As a result of remnant configuration, the semantic constraints defined in the data model cannot be relied upon for <operational>, since the system may have remnant configuration whose constraints were valid with the previous configuration and that are not valid with the current configuration. Since constraints on "config false" nodes may refer to "config true" nodes, remnant configuration may force the violation of those constraints. The constraints that may not hold include "when", "must", "min-elements", and "max-elements". Note that syntactic constraints cannot be violated, including hierarchical organization, identifiers, and type-based constraints.

4.7.1. **Missing Resources**

Configuration in <intended> can refer to resources that are not available or otherwise not physically present. In these situations, these parts of the <intended> configuration are not applied. The data appears in <intended> but does not appear in <operational>.

A typical example is an interface configuration that refers to an interface that is not currently present. In such a situation, the interface configuration remains in <intended> but the interface configuration will not appear in <operational>.

Note that configuration validity cannot depend on the current state of such resources, since that would imply the removing a resource might render the configuration invalid. This is unacceptable, especially given that rebooting such a device would fail to boot due to an invalid configuration. Instead we allow configuration for missing resources to exist in <running> and <intended>, but it will not appear in <operational>.

4.7.2. System-controlled Resources

Sometimes resources are controlled by the device and the corresponding system controlled data appear in (and disappear from) <operational> dynamically. If a system controlled resource has matching configuration in <intended> when it appears, the system will try to apply the configuration, which causes the configuration to appear in <operational> eventually (if application of the configuration was successful).

4.7.3. Origin Metadata Annotation

As data flows into <operational>, it is conceptually marked with a metadata annotation ([RFC7952]) that indicates its origin. The origin applies to all data nodes except non-presence containers. The "origin" metadata annotation is defined in Section 6. The values are YANG identities. The following identities are defined:

- origin: abstract base identity from which the other origin identities are derived.

- intended: represents data provided by <intended>.

- dynamic: represents data provided by a dynamic datastore.

- system: represents data provided by the system itself, including both system configuration and system state. Examples of system configuration include applied configuration for an always existing loopback interface, or interface configuration that is auto-created due to the hardware currently present in the device.

- learned: represents configuration that has been learned via protocol interactions with other systems, including protocols such as link-layer negotiations, routing protocols, DHCP, etc.

- default: represents data using a default value specified in the data model, using either values in the "default" statement or any values described in the "description" statement. The default origin is only used when the data has not been provided by any other source.

- unknown: represents data for which the system cannot identify the origin.

These identities can be further refined, e.g., there could be separate identities for particular types or instances of dynamic datastore derived from "dynamic".

In all cases, the device should report the origin that most accurately reflects the source of the data that is actively being used by the system.

In cases where it could be ambiguous as to which origin should be used, i.e. where the same data node value has originated from multiple sources, then the description statement in the YANG module should be used as guidance for choosing the appropriate origin. For example:

If for a particular configuration node, the associated YANG description statement indicates that a protocol negotiated value overrides any configured value, then the origin would be reported as "learned", even when a learned value is the same as the configured value.

Conversely, if for a particular configuration node, the associated YANG description statement indicates that a protocol negotiated value does not override an explicitly configured value, then the origin would be reported as "intended" even when a learned value is the same as the configured value.

In the case that a device cannot provide an accurate origin for a particular data node then it should use the origin "unknown".

5. Implications on YANG

5.1. XPath Context

If a server implements the architecture defined in this document, the accessible trees for some XPath contexts are refined as follows:

- If the XPath expression is defined in a substatement to a data node that represents system state, the accessible tree is all operational state in the server. The root node has all top-level data nodes in all modules as children.

- If the XPath expression is defined in a substatement to a "notification" statement, the accessible tree is the notification instance and all operational state in the server. If the notification is defined on the top level in a module, then the

   root node has the node representing the notification being defined and all top-level data nodes in all modules as children. Otherwise, the root node has all top-level data nodes in all modules as children.

o If the XPath expression is defined in a substatement to an "input" statement in an "rpc" or "action" statement, the accessible tree is the RPC or action operation instance and all operational state in the server. The root node has top-level data nodes in all modules as children. Additionally, for an RPC, the root node also has the node representing the RPC operation being defined as a child. The node representing the operation being defined has the operation's input parameters as children.

o If the XPath expression is defined in a substatement to an "output" statement in an "rpc" or "action" statement, the accessible tree is the RPC or action operation instance and all operational state in the server. The root node has top-level data nodes in all modules as children. Additionally, for an RPC, the root node also has the node representing the RPC operation being defined as a child. The node representing the operation being defined has the operation's output parameters as children.

6. YANG Modules

```
<CODE BEGINS> file "ietf-datastores@2017-04-26.yang"

module ietf-datastores {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-datastores";
  prefix ds;

  organization
    "IETF NETMOD (NETCONF Data Modeling Language) Working Group";

  contact
    "WG Web:   <https://datatracker.ietf.org/wg/netmod/>

     WG List:  <mailto:netmod@ietf.org>

     Author:   Martin Bjorklund
               <mailto:mbj@tail-f.com>

     Author:   Juergen Schoenwaelder
               <mailto:j.schoenwaelder@jacobs-university.de>

     Author:   Phil Shafer
               <mailto:phil@juniper.net>
```

```
        Author:   Kent Watsen
                  <mailto:kwatsen@juniper.net>

        Author:   Rob Wilton
                  <rwilton@cisco.com>";

     description
       "This YANG module defines two sets of identities for datastores.
        The first identifies the datastores themselves, the second
        identifies are for datastore protperties.

        Copyright (c) 2017 IETF Trust and the persons identified as
        authors of the code.  All rights reserved.

        Redistribution and use in source and binary forms, with or
        without modification, is permitted pursuant to, and subject to
        the license terms contained in, the Simplified BSD License set
        forth in Section 4.c of the IETF Trust's Legal Provisions
        Relating to IETF Documents
        (http://trustee.ietf.org/license-info).

        This version of this YANG module is part of RFC XXXX
        (http://www.rfc-editor.org/info/rfcxxxx); see the RFC itself
        for full legal notices.";

     revision 2017-04-26 {
       description
         "Initial revision.";
       reference
         "RFC XXXX: Network Management Datastore Architecture";
     }

     /*
      * Identities
      */

     identity datastore {
       description
        "Abstract base identity for datastore identities.";
     }

     identity conventional {
       base datastore;
       description
        "Abstract base identity for conventional configuration
         datastores.";
     }
```

```
     identity running {
       base conventional;
       description
        "The running configuration datastore.";
     }

     identity candidate {
       base conventional;
       description
        "The candidate configuration datastore.";
     }

     identity startup {
       base conventional;
       description
        "The startup configuration datastore.";
     }

     identity intended {
       base conventional;
       description
        "The intended configuration datastore.";
     }

     identity dynamic {
       base datastore;
       description
        "Abstract base identity for dynamic datastores.";
     }

     identity operational {
       base datastore;
       description
        "The operational state datastore.";
     }



     identity property {
       description
        "Abstract base identity for datastore identities.";
     }

     identity writable {
       base property;
       description
         "Used on the 'running' datastore to indicate that it can be
          written to.";
```

```
     }

     identity auto-persist {
       base property;
       description
         "Used on the 'running' datastore to indicate that writes to
          it will be automatically persisted.

          If the 'startup' datastore is also supported, clients may
          query its contents to ensure its synchronization.

          If the 'startup' datastore is not supported, and this
          property is not set, then clients must use a mechanism
          provided by the protocol to explicitly persist the
          'running' datastore's contents.";
     }

     identity rollback-on-error {
       base property;
       description
         "Used on either the 'running' or 'candidate' datastores to
          indicate that clients may request atomic update behavior.";
     }

     identity confirmed-commit {
       base property;
       description
         "Used on the 'candidate' datastore to indicate that
          clients may request confirmed-commit update behavior.";
     }

     identity validate {
       base property;
       description
         "Used on the 'candidate' datastore to indicate that
          clients may request datastore validation.";
     }

   }


   <CODE ENDS>

   <CODE BEGINS> file "ietf-origin@2017-04-26.yang"

   module ietf-origin {
     yang-version 1.1;
     namespace "urn:ietf:params:xml:ns:yang:ietf-origin";
```

```
     prefix or;

     import ietf-yang-metadata {
       prefix md;
     }

     organization
       "IETF NETMOD (NETCONF Data Modeling Language) Working Group";

     contact
       "WG Web:   <https://datatracker.ietf.org/wg/netmod/>

        WG List:  <mailto:netmod@ietf.org>

        Author:   Martin Bjorklund
                  <mailto:mbj@tail-f.com>

        Author:   Juergen Schoenwaelder
                  <mailto:j.schoenwaelder@jacobs-university.de>

        Author:   Phil Shafer
                  <mailto:phil@juniper.net>

        Author:   Kent Watsen
                  <mailto:kwatsen@juniper.net>

        Author:   Rob Wilton
                  <rwilton@cisco.com>";

     description
       "This YANG module defines an 'origin' metadata annotation, and a
        set of identities for the origin value.

        Copyright (c) 2017 IETF Trust and the persons identified as
        authors of the code.  All rights reserved.

        Redistribution and use in source and binary forms, with or
        without modification, is permitted pursuant to, and subject to
        the license terms contained in, the Simplified BSD License set
        forth in Section 4.c of the IETF Trust's Legal Provisions
        Relating to IETF Documents
        (http://trustee.ietf.org/license-info).

        This version of this YANG module is part of RFC XXXX
        (http://www.rfc-editor.org/info/rfcxxxx); see the RFC itself
        for full legal notices.";

     revision 2017-04-26 {
```

```
       description
         "Initial revision.";
       reference
         "RFC XXXX: Network Management Datastore Architecture";
     }

     /*
      * Identities
      */

     identity origin {
       description
         "Abstract base identity for the origin annotation.";
     }

     identity intended {
       base origin;
       description
         "Denotes data from the intended configuration datastore";
     }

     identity dynamic {
       base origin;
       description
         "Denotes data from a dynamic datastore.";
     }

     identity system {
       base origin;
       description
         "Denotes data originated by the system itself, including
          both system configuration and system state.

          Examples of system configuration include applied configuration
          for an always existing loopback interface, or interface
          configuration that is auto-created due to the hardware
          currently present in the device.";
     }

     identity learned {
       base origin;
       description
         "Denotes configuration learned from protocol interactions with
          other devices, instead of via the intended configuration
          datastore or any dynamic datastore.

          Examples of protocols that provide learned configuration
          include link-layer negotiations, routing protocols, and
```

```
          DHCP.";
     }

     identity default {
       base origin;
       description
         "Denotes data that does not have an configured or learned
          value, but has a default value in use.  Covers both values
          defined in a 'default' statement, and values defined via an
          explanation in a 'description' statement.";
     }

     identity unknown {
       base origin;
       description
         "Denotes data for which the system cannot identify the
          origin.";
     }

     /*
      * Metadata annotations
      */

     md:annotation origin {
       type identityref {
         base origin;
       }
       description
         "The 'origin' annotation can be present on any node in a
          datastore.  It specifies from where the node originated.";
     }

   }

   <CODE ENDS>
```

7. IANA Considerations

7.1. Updates to the IETF XML Registry

This document registers two URIs in the IETF XML registry [RFC3688]. Following the format in [RFC3688], the following registrations are requested:

```
      URI: urn:ietf:params:xml:ns:yang:ietf-datastores
      Registrant Contact: The IESG.
      XML: N/A, the requested URI is an XML namespace.

      URI: urn:ietf:params:xml:ns:yang:ietf-origin
      Registrant Contact: The IESG.
      XML: N/A, the requested URI is an XML namespace.
```

7.2. Updates to the YANG Module Names Registry

This document registers two YANG modules in the YANG Module Names registry [RFC6020]. Following the format in [RFC6020], the the following registrations are requested:

```
      name:         ietf-datastores
      namespace:    urn:ietf:params:xml:ns:yang:ietf-datastores
      prefix:       ds
      reference:    RFC XXXX

      name:         ietf-origin
      namespace:    urn:ietf:params:xml:ns:yang:ietf-origin
      prefix:       or
      reference:    RFC XXXX
```

8. Security Considerations

This document discusses an architectural model of datastores for network management using NETCONF/RESTCONF and YANG. It has no security impact on the Internet.

9. Acknowledgments

This document grew out of many discussions that took place since 2010. Several Internet-Drafts ([I-D.bjorklund-netmod-operational], [I-D.wilton-netmod-opstate-yang], [I-D.ietf-netmod-opstate-reqs], [I-D.kwatsen-netmod-opstate], [I-D.openconfig-netmod-opstate]) and [RFC6244] touched on some of the problems of the original datastore model. The following people were authors to these Internet-Drafts or otherwise actively involved in the discussions that led to this document:

o Lou Berger, LabN Consulting, L.L.C., <lberger@labn.net>

o Andy Bierman, YumaWorks, <andy@yumaworks.com>

o Marcus Hines, Google, <hines@google.com>

o Christian Hopps, Deutsche Telekom, <chopps@chopps.org>

o Acee Lindem, Cisco Systems, <acee@cisco.com>

o Ladislav Lhotka, CZ.NIC, <lhotka@nic.cz>

o Thomas Nadeau, Brocade Networks, <tnadeau@lucidvision.com>

o Anees Shaikh, Google, <aashaikh@google.com>

o Rob Shakir, Google, <robjs@google.com>

Juergen Schoenwaelder was partly funded by Flamingo, a Network of Excellence project (ICT-318488) supported by the European Commission under its Seventh Framework Programme.

10. References

10.1. Normative References

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <http://www.rfc-editor.org/info/rfc6241>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <http://www.rfc-editor.org/info/rfc7950>.

[RFC7952] Lhotka, L., "Defining and Using Metadata with YANG", RFC 7952, DOI 10.17487/RFC7952, August 2016, <http://www.rfc-editor.org/info/rfc7952>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <http://www.rfc-editor.org/info/rfc8040>.

10.2. Informative References

[I-D.bjorklund-netmod-operational]
Bjorklund, M. and L. Lhotka, "Operational Data in NETCONF and YANG", draft-bjorklund-netmod-operational-00 (work in progress), October 2012.

[I-D.ietf-netmod-opstate-reqs]
Watsen, K. and T. Nadeau, "Terminology and Requirements for Enhanced Handling of Operational State", draft-ietf-netmod-opstate-reqs-04 (work in progress), January 2016.

[I-D.kwatsen-netmod-opstate]
Watsen, K., Bierman, A., Bjorklund, M., and J. Schoenwaelder, "Operational State Enhancements for YANG, NETCONF, and RESTCONF", draft-kwatsen-netmod-opstate-02 (work in progress), February 2016.

[I-D.openconfig-netmod-opstate]
Shakir, R., Shaikh, A., and M. Hines, "Consistent Modeling of Operational State Data in YANG", draft-openconfig-netmod-opstate-01 (work in progress), July 2015.

[I-D.wilton-netmod-opstate-yang]
Wilton, R., ""With-config-state" Capability for NETCONF/RESTCONF", draft-wilton-netmod-opstate-yang-02 (work in progress), December 2015.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <http://www.rfc-editor.org/info/rfc3688>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <http://www.rfc-editor.org/info/rfc6020>.

[RFC6244] Shafer, P., "An Architecture for Network Management Using NETCONF and YANG", RFC 6244, DOI 10.17487/RFC6244, June 2011, <http://www.rfc-editor.org/info/rfc6244>.

Appendix A. Guidelines for Defining Datastores

The definition of a new datastore in this architecture should be provided in a document (e.g., an RFC) purposed to the definition of the datastore. When it makes sense, more than one datastore may be defined in the same document (e.g., when the datastores are logically connected). Each datastore's definition should address the points specified in the sections below.

A.1. Define which YANG modules can be used in the datastore

Not all YANG modules may be used in all datastores. Some datastores may constrain which data models can be used in them. If it is desirable that a subset of all modules can be targeted to the datastore, then the documentation defining the datastore must indicate this.

A.2. Define which subset of YANG-modeled data applies

By default, the data in a datastore is modeled by all YANG statements in the available YANG modules. However, it is possible to specify criteria that YANG statements must satisfy in order to be present in a datastore. For instance, maybe only "config true" nodes are present, or "config false nodes" that also have a specific YANG extension (e.g., "i2rs:ephemeral true") are present in the datastore.

A.3. Define how data is actualized

The new datastore must specify how it interacts with other datastores. For example, the diagram in Section 4 depicts dynamic datastores feeding into <operational>. How this interaction occurs must be defined by any dynamic datastore. In some cases, it may occur implicitly, as soon as the data is put into the dynamic datastore while, in other cases, an explicit action (e.g., an RPC) may be required to trigger the application of the datastore's data.

A.4. Define which protocols can be used

By default, it is assumed that both the NETCONF and RESTCONF protocols can be used to interact with a datastore. However, it may be that only a specific protocol can be used (e.g., ForCES) or that a subset of all protocol operations or capabilities are available (e.g., no locking or no XPath-based filtering).

A.5. Define YANG identities for the datastore

The datastore must be defined with a YANG identity that uses the "ds:datastore" identity or one of its derived identities as its base. This identity is necessary so that the datastore can be referenced in protocol operations (e.g., <get-data>).

The datastore may also be defined with an identity that uses the "or:origin" identity or one its derived identities as its base. This identity is needed if the datastore interacts with <operational> so that data originating from the datastore can be identified as such via the "origin" metadata attribute defined in Section 6.

An example of these guidelines in use is provided in Appendix B.

Appendix B. Ephemeral Dynamic Datastore Example

The section defines documentation for an example dynamic datastore using the guidelines provided in Appendix A. While this example is very terse, it is expected to be that a standalone RFC would be needed when fully expanded.

This example defines a dynamic datastore called "ephemeral", which is loosely modeled after the work done in the I2RS working group.

```
  1. Name             : ephemeral
  2. YANG modules     : all (default)
  3. YANG statements  : config false + ephemeral true
  4. How applied      : automatic
  5. Protocols        : NC/RC (default)
  6. YANG Module      : (see below)
```

```
module example-ds-ephemeral {
  yang-version 1.1;
  namespace "urn:example:ds-ephemeral";
  prefix eph;

  import ietf-datastores {
    prefix ds;
  }
  import ietf-origin {
    prefix or;
  }

  // add datastore identity
  identity ds-ephemeral {
    base ds:datastore;
    description
      "The 'ephemeral' datastore.";
  }

  // add origin identity
  identity or-ephemeral {
    base or:dynamic;
    description
      "Denotes data from the ephemeral dynamic datastore.";
  }

  // define ephemeral extension
  extension ephemeral {
    argument "value";
    description
      "This extension is mixed into config false YANG nodes to
       indicate that they are writable nodes in the 'ephemeral'
       datastore.  This statement takes a single argument
       representing a boolean having the values 'true' and
       'false'.  The default value is 'false'.";
  }
}
```

Appendix C. Example Data

The use of datastores is complex, and many of the subtle effects are more easily presented using examples. This section presents a series of example data models with some sample contents of the various datastores.

C.1. System Example

In this example, the following fictional module is used:

```
module example-system {
  yang-version 1.1;
  namespace urn:example:system;
  prefix sys;

  import ietf-inet-types {
    prefix inet;
  }

  container system {
    leaf hostname {
      type string;
    }

    list interface {
      key name;

      leaf name {
        type string;
      }

      container auto-negotiation {
        leaf enabled {
          type boolean;
          default true;
        }
        leaf speed {
          type uint32;
          units mbps;
          description
            "The advertised speed, in mbps.";
        }
      }

      leaf speed {
        type uint32;
        units mbps;
```

```
           config false;
           description
             "The speed of the interface, in mbps.";
         }

         list address {
           key ip;

           leaf ip {
             type inet:ip-address;
           }
           leaf prefix-length {
             type uint8;
           }
         }
       }
     }
   }
```

The operator has configured the host name and two interfaces, so the contents of <intended> is:

```
   <system xmlns="urn:example:system">

     <hostname>foo</hostname>

     <interface>
       <name>eth0</name>
       <auto-negotiation>
         <speed>1000</speed>
       </auto-negotiation>
       <address>
         <ip>2001:db8::10</ip>
         <prefix-length>32</prefix-length>
       </address>
     </interface>

     <interface>
       <name>eth1</name>
       <address>
         <ip>2001:db8::20</ip>
         <prefix-length>32</prefix-length>
       </address>
     </interface>

   </system>
```

The system has detected that the hardware for one of the configured interfaces ("eth1") is not yet present, so the configuration for that interface is not applied. Further, the system has received a host name and an additional IP address for "eth0" over DHCP. In addition to a default value, a loopback interface is automatically added by the system, and the result of the "speed" auto-negotiation. All of this is reflected in <operational>:

```
<system
    xmlns="urn:example:system"
    xmlns:or="urn:ietf:params:xml:ns:yang:ietf-origin">

  <hostname or:origin="or:dynamic">bar</hostname>

  <interface or:origin="or:intended">
    <name>eth0</name>
    <auto-negotiation>
      <enabled or:origin="or:default">true</enabled>
      <speed>1000</speed>
    </auto-negotiation>
    <speed>100</speed>
    <address>
      <ip>2001:db8::10</ip>
      <prefix-length>64</prefix-length>
    </address>
    <address or:origin="or:dynamic">
      <ip>2001:db8::1:100</ip>
      <prefix-length>64</prefix-length>
    </address>
  </interface>

  <interface or:origin="or:system">
    <name>lo0</name>
    <address>
      <ip>::1</ip>
      <prefix-length>128</prefix-length>
    </address>
  </interface>

</system>
```

C.2. BGP Example

Consider the following piece of a ersatz BGP module:

```
       container bgp {
         leaf local-as {
           type uint32;
         }
         leaf peer-as {
           type uint32;
         }
         list peer {
           key name;
           leaf name {
             type ipaddress;
           }
           leaf local-as {
             type uint32;
             description
               ".... Defaults to ../local-as";
           }
           leaf peer-as {
             type uint32;
             description
                "... Defaults to ../peer-as";
           }
           leaf local-port {
             type inet:port;
           }
           leaf remote-port {
             type inet:port;
             default 179;
           }
           leaf state {
             config false;
             type enumeration {
               enum init;
               enum established;
               enum closing;
             }
           }
         }
       }
```

In this example model, both bgp/peer/local-as and bgp/peer/peer-as have complex hierarchical values, allowing the user to specify default values for all peers in a single location.

The model also follows the pattern of fully integrating state ("config false") nodes with configuration ("config true") nodes. There is not separate "bgp-state" hierarchy, with the accompanying

repetition of containment and naming nodes. This makes the model simpler and more readable.

C.2.1. Datastores

Each datastore represents differing views of these nodes. <running> will hold the configuration provided by the user, for example a single BGP peer. <intended> will conceptually hold the data as validated, after the removal of data not intended for validation and after any local template mechanisms are performed. <operational> will show data from <intended> as well as any "config false" nodes.

C.2.2. Adding a Peer

If the user configures a single BGP peer, then that peer will be visible in both <running> and <intended>. It may also appear in <candidate>, if the server supports the "candidate" feature. Retrieving the peer will return only the user-specified values.

No time delay should exist between the appearance of the peer in <running> and <intended>.

In this scenario, we've added the following to <running>:

```
<bgp>
  <local-as>64642</local-as>
  <peer-as>65000</peer-as>
  <peer>
    <name>10.1.2.3</name>
  </peer>
</bgp>
```

C.2.2.1. <operational>

<operational> will contain the fully expanded peer data, including "config false" nodes. In our example, this means the "state" node will appear.

In addition, <operational> will contain the "currently in use" values for all nodes. This means that local-as and peer-as will be populated even if they are not given values in <intended>. The value of bgp/local-as will be used if bgp/peer/local-as is not provided; bgp/peer-as and bgp/peer/peer-as will have the same relationship. In the operational view, this means that every peer will have values for their local-as and peer-as, even if those values are not explicitly configured but are provided by bgp/local-as and bgp/peer-as.

Each BGP peer has a TCP connection associated with it, using the values of local-port and remote-port from <intended>. If those values are not supplied, the system will select values. When the connection is established, <operational> will contain the current values for the local-port and remote-port nodes regardless of the origin. If the system has chosen the values, the "origin" attribute will be set to "system". Before the connection is established, one or both of the nodes may not appear, since the system may not yet have their values.

```
<bgp origin="or:intended" xmlns="urn:example:bgp">
  <local-as origin="or:intended">64642</local-as>
  <peer-as origin="or:intended">65000</peer-as>
  <peer origin="or:intended">
    <name origin="or:intended">10.1.2.3</name>
    <local-as origin="or:default">64642</local-as>
    <peer-as origin="or:default">65000</peer-as>
    <local-port origin="or:system">60794</local-port>
    <remote-port origin="or:default">179</remote-port>
  </peer>
</bgp>
```

C.2.3. Removing a Peer

Changes to configuration may take time to percolate through the various software components involved. During this period, it is imperative to continue to give an accurate view of the working of the device. <operational> will contain nodes for both the previous and current configuration, as closely as possible tracking the current operation of the device.

Consider the scenario where a client removes a BGP peer. When a peer is removed, the operational state will continue to reflect the existence of that peer until the peer's resources are released, including closing the peer's connection. During this period, the current data values will continue to be visible in <operational>, with the "origin" attribute set to indicate the origin of the original data.

```
     <bgp origin="or:intended">
       <local-as origin="or:intended">64642</local-as>
       <peer-as origin="or:intended">65000</peer-as>
       <peer origin="or:intended">
         <name origin="or:intended">10.1.2.3</name>
         <local-as origin="or:default">64642</local-as>
         <peer-as origin="or:default">65000</peer-as>
         <local-port origin="or:system">60794</local-port>
         <remote-port origin="or:default">179</remote-port>
       </peer>
     </bgp>
```

Once resources are released and the connection is closed, the peer's data is removed from <operational>.

C.3. Interface Example

In this section, we'll use this simple interface data model:

```
     container interfaces {
       list interface {
         key name;
         leaf name {
           type string;
         }
         leaf description {
           type string;
         }
         leaf mtu {
           type uint;
         }
         leaf ipv4-address {
           type inet:ipv4-address;
         }
       }
     }
```

C.3.1. Pre-provisioned Interfaces

One common issue in networking devices is the support of Field Replaceable Units (FRUs) that can be inserted and removed from the device without requiring a reboot or interfering with normal operation. These FRUs are typically interface cards, and the devices support pre-provisioning of these interfaces.

If a client creates an interface "et-0/0/0" but the interface does not physically exist at this point, then <intended> might contain the following:

```
     <interfaces>
       <interface>
         <name>et-0/0/0</name>
         <description>Test interface</description>
       </interface>
     </interfaces>
```

Since the interface does not exist, this data does not appear in <operational>.

When a FRU containing this interface is inserted, the system will detect it and process the associated configuration. The <operational> will contain the data from <intended>, as well as the "config false" nodes, such as the current value of the interface's MTU.

```
     <interfaces origin="or:intended">
       <interface origin="or:intended">
         <name origin="or:intended">et-0/0/0</name>
         <description origin="or:intended">Test interface</description>
         <mtu origin="or:system">1500</mtu>
       </interface>
     </interfaces>
```

If the FRU is removed, the interface data is removed from <operational>.

C.3.2. System-provided Interface

Imagine if the system provides a loopback interface (named "lo0") with a default ipv4-address of "127.0.0.1". The system will only provide configuration for this interface if there is no data for it in <intended>.

When no configuration for "lo0" appears in <intended>, then <operational> will show the system-provided data:

```
     <interfaces origin="or:intended">
       <interface origin="or:system">
         <name origin="or:system">lo0</name>
         <ipv4-address origin="or:system">127.0.0.1</ipv4-address>
       </interface>
     </interfaces>
```

When configuration for "lo0" does appear in <intended>, then <operational> will show that data with the origin set to "intended". If the "ipv4-address" is not provided, then the system-provided value will appear as follows:

```
     <interfaces origin="or:intended">
       <interface origin="or:intended">
         <name origin="or:intended">lo0</name>
         <description origin="or:intended">loopback</description>
         <ipv4-address origin="or:system">127.0.0.1</ipv4-address>
       </interface>
     </interfaces>
```

Authors' Addresses

Martin Bjorklund
Tail-f Systems

Email: mbj@tail-f.com

Juergen Schoenwaelder
Jacobs University

Email: j.schoenwaelder@jacobs-university.de

Phil Shafer
Juniper Networks

Email: phil@juniper.net

Kent Watsen
Juniper Networks

Email: kwatsen@juniper.net

Robert Wilton
Cisco Systems

Email: rwilton@cisco.com