Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Bjorklund
Tail-f Systems
J. Schoenwaelder
Jacobs University
P. Shafer
K. Watsen
Juniper Networks
R. Wilton
Cisco Systems
March 13, 2017

Network Management Datastore Architecture
draft-ietf-netmod-revised-datastores-01

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

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].

This document defines the following terms:

- o configuration data: Data that determines how a device behaves. This data is modeled in YANG using "config true" nodes. Configuration data can originate from different sources.

- static configuration data: Configuration data that is eventually persistent and used to get a device from its initial default state into its desired operational state.

- dynamic configuration data: Configuration data that is obtained dynamically during the operation of a device through interaction with other systems and not persistent.

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

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

- applied configuration data: Configuration data that is currently used by a device. Applied configuration data consists of static configuration data and dynamic configuration data.

- state data: The additional data on a system that is not configuration data such as read-only status information and collected statistics. State data is transient and modified by interactions with internal components or other systems. State data is modeled in YANG using "config false" nodes.

- 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.

- configuration datastore: A datastore holding static configuration data that is required to get a device from its initial default state into a desired operational state. A configuration datastore maps to an instantiated YANG data tree consisting of configuration data nodes and interior data nodes.

- running configuration datastore: A configuration datastore holding the complete static configuration currently active on the device. The running configuration datastore always exists. It may include inactive configuration or template-mechanism-oriented configuration that require further expansion.

- intended configuration datastore: A configuration datastore holding the complete configuration currently active on the device. It does not include inactive configuration and it does include the expansion of any template mechanisms.

- 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. A candidate datastore may not be supported by all protocols or implementations.

- startup configuration datastore: The configuration datastore holding the configuration loaded by the device into the running configuration datastore when it boots. A startup datastore may not be supported by all protocols or implementations.

- dynamic datastore: A datastore holding dynamic configuration data.

- operational state datastore: A datastore holding the currently active applied configuration data as well as the device's state data.

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

- remnant data: Configuration data that remains in the system for a period of time after it has be removed from a configuration datastore. The time period may be minimal, or may last until all resources used by the newly-deleted configuration data (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:

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

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

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

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

3. Introduction

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 data 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 did exist):

- 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 data.

[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 a separate data tree 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 data from configuration data 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 data and on related operational state data 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 the <candidate> and <startup> datastores is optional and the <running> datastore does not have to be writable. Furthermore, the <startup> datastore 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 a <candidate> datastore or by directly modifying the <running> datastore 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 the <running> datastore. NETCONF explicitly mentions so called named datastores.

Some observations:

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

o 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 data can have a different lifetime compared to configuration data or if configuration data is not immediately or successfully applied.

- Several implementations have proprietary mechanisms that allow clients to store inactive data in the <running> datastore; 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 the <running> datastore 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)  |
                      +-----------+
                            |
                            |        // e.g., removal of "inactive"
                            |        // nodes, expansion of templates
                            v
                      +------------+
                      | <intended> | // subject to validation
                      | (ct, ro)   |
                      +------------+
                            |
                            |        // e.g., missing resources, delays
                            |
                            |   +------ auto-discovery
                            |   +------ dynamic configuration protocols
                            |   +------ control-plane protocols
                            |   +------ dynamic datastores
                            |   |
                            v   v
                    +---------------+
                    | <operational> |
                    | (ct + cf, ro) |
                    +---------------+

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

4.1. The <intended> Datastore

The <intended> datastore is a read-only datastore that consists of config true nodes. 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>.

On a traditional NETCONF implementation, <running> and <intended> are always the same.

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.

4.2. 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 the <operational> datastore.

Note that the ephemeral datastore discussed in I2RS documents maps to a dynamic datastore in the datastore model described here.

4.3. The <operational> Datastore

The <operational> datastore is a read-only datastore that consists of config true and config false nodes. 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.

The <operational> datastore contains all configuration data actually used by the system, including all applied configuration, system-provided configuration and values defined by any supported data models. In addition, the <operational> datastore also contains state data.

Changes to configuration data may take time to percolate through to the <operational> datastore. During this period, the <operational> datastore will return data nodes for both the previous and current configuration, as closely as possible tracking the current operation of the device. These "remnants" of the previous configuration persist while the system has released resources used by the newly-deleted configuration data (e.g., network connections, memory allocations, file handles).

As a result of these remnants, the semantic constraints defined in the data model cannot be relied upon for the <operational> datastore, since the system may have remnants 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, remnants 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.

4.3.1. Missing Resources

The <intended> configuration 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>.

4.3.2. 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).

4.3.3. Origin Metadata Annotation

As data flows into the <operational> datastore, it is conceptually marked with a metadata annotation ([RFC7952]) that indicates its origin. The "origin" metadata annotation is defined in Section 6. The values are YANG identities. The following identities are defined:

```
+-- origin
    +-- static
    +-- dynamic
    +-- default
    +-- system
```

These identities can be further refined, e.g., there might be an identity "dhcp" derived from "dynamic".

The "static" origin represents data provided by the <intended> datastore. The "dynamic" origin represents data provided by a dynamic datastore. The "default" origin represents data values specified in the data model, using either simple values in the "default" statement or any values described in the "description" statement. Finally, the "system" origin represents data learned from the normal operational of the system, including control-plane protocols.

5. Guidelines for Defining Dynamic Datastores

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

5.1. Define a name for the dynamic datastore

Each dynamic datastores MUST have a name using the character set described by Section 6.2 of [RFC7950]. The name SHOULD be consistent in style and length to other datastore names described in this document.

The datastore’s name does not need to be globally unique, as it will be uniquely qualified by the namespace of the module in which it is defined (Section 5.6). This means that names such as "running" and "operational" are valid datastore names. However, it is usually desirable to avoid using the same name for multiple different datastores.

5.2. 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 dynamic datastore, then the documentation defining the dynamic datastore MUST use the mechanisms described in Appendix D.2 to provide the necessary hooks for module-designers to indicate that their module is to be accessible in the dynamic datastore.

5.3. Define which subset of YANG-modeled data applies

By default, the data in a dynamic datastore is modeled by all YANG statements in the available YANG modules. However, it is possible to specify criteria YANG statements must satisfy in order to be present in a dynamic 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 dynamic datastore.

5.4. Define how dynamic data is actualized

The diagram in Section 4 depicts dynamic datastores feeding into the <operational> datastore. How this interaction occurs must be defined by the 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 dynamic datastore’s data.

5.5. 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 dynamic 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, no xpath-based filtering, etc.).

5.6. Define a module for the dynamic datastore

Each dynamic datastore MUST be defined by a YANG module. This module is used by servers to indicate (e.g., via YANG Library) their support for the dynamic datastore.

The YANG module MUST import the "ietf-datastores" and "ietf-origin" modules, defined in this document. This is necessary in order to access the base identities they define.

The YANG module MUST define an identity that uses the "ds:datastore" identity as its base. This identity is necessary so that the datastore can be referenced in protocol operations (e.g., <get-data>).

The YANG module MUST define an identity that uses the "or:dynamic" identity as its base. This identity is necessary 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.

6. YANG Modules

```
<CODE BEGINS> file "ietf-datastores@2017-03-13.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 a set of identities for datastores.
     These identities can be used to identify datastores in protocol
     operations.

     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 2017-03-13 {
       description
         "Initial revision.";
       reference
         "RFC XXXX: Network Management Datastore Architecture";
     }

     /*
      * Identities
      */

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

     identity static {
       description
        "Abstract base identity for static configuration datastores.";
     }

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

     identity running {
       base static;
       description
        "The 'running' datastore.";
     }

     identity candidate {
       base static;
       description
        "The 'candidate' datastore.";
     }

     identity startup {
       base static;
```

```
       description
        "The 'startup' datastore.";
     }

     identity intended {
       base static;
       description
        "The 'intended' datastore.";
     }

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

   }

   <CODE ENDS>

   <CODE BEGINS> file "ietf-datastores@2017-03-13.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.  The 'origin' metadata
        annotation is used to mark data in the 'operational'
        datastore with information on where the data originated.

        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 2017-03-13 {
       description
         "Initial revision.";
       reference
         "RFC XXXX: Network Management Datastore Architecture";
     }

     /*
      * Identities
      */

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

     identity static {
       base origin;
       description
         "Denotes data from static configuration (e.g., <intended>).";
     }
```

```
     identity dynamic {
       base origin;
       description
         "Denotes data from dynamic configuration protocols
          or dynamic datastores (e.g., DHCP).";
     }

     identity system {
       base origin;
       description
         "Denotes data created by the system independently of what
          has been configured.";
     }

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

     /*
      * Metadata annotations
      */

     md:annotation origin {
       type identityref {
         base origin;
       }
     }

   }

   <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 a conceptual 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:

- Lou Berger, LabN Consulting, L.L.C., <lberger@labn.net>
- Andy Bierman, YumaWorks, <andy@yumaworks.com>
- Marcus Hines, Google, <hines@google.com>
- 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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[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>.

[RFC7895] Bierman, A., Bjorklund, M., and K. Watsen, "YANG Module Library", RFC 7895, DOI 10.17487/RFC7895, June 2016, <http://www.rfc-editor.org/info/rfc7895>.

[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.ietf-netmod-rfc6087bis]
Bierman, A., "Guidelines for Authors and Reviewers of YANG Data Model Documents", draft-ietf-netmod-rfc6087bis-12 (work in progress), March 2017.

[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>.

[RFC6243] Bierman, A. and B. Lengyel, "With-defaults Capability for NETCONF", RFC 6243, DOI 10.17487/RFC6243, June 2011, <http://www.rfc-editor.org/info/rfc6243>.

[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. 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.

A.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:static">
    <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>32</prefix-length>
    </address>
    <address or:origin="or:dynamic">
      <ip>2001:db8::1:100</ip>
      <prefix-length>32</prefix-length>
    </address>
  </interface>

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

</system>
```

A.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.

A.2.1. Datastores

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

A.2.2. Adding a Peer

If the user configures a single BGP peer, then that peer will be visible in both the <running> and <intended> datastores. It may also appear in the <candidate> datastore, 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>
```

A.2.2.1. <operational>

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

In addition, the <operational> datastore 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 the intended datastore. If those values are not supplied, the system will select values. When the connection is established, the <operational> datastore 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 "operational". 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:static" xmlns="urn:example:bgp">
  <local-as origin="or:static">64642</local-as>
  <peer-as origin="or:static">65000</peer-as>
  <peer origin="or:static">
    <name origin="or:static">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>
```

A.2.3. Removing a Peer

Changes to configuration data 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. The <operational> datastore will return data 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 the <operational> datastore, with the "origin" attribute set to indicate the origin of the original data.

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

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

A.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;
    }
  }
}
```

A.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 the <intended> datastore 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 the <operational> datastore.

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

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

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

Appendix B. Ephemeral Dynamic Datastore Example

The section defines documentation for an example dynamic datastore using the guidelines provided in Section 5. 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. Implications on Data Models

Since the NETCONF <get/> operation returns the content of the <running> configuration datastore and the operational state together in one tree, data models were often forced to branch at the top-level into a config true branch and a structurally similar config false branch that replicated some of the config true nodes and added state nodes. With the datastore model described here this is not needed anymore since the different datastores handle the different lifetimes of data objects. Introducing this model together with the deprecation of the <get/> operation makes it possible to write simpler models.

C.1. Proposed migration of existing YANG Data Models

For standards based YANG modules that have already been published, that are using split config and state trees, it is planned that these modules are updated with new revisions containing the following changes:

- o The top level module description is updated to indicate that the module conforms to the revised datastore architecture with a combined config and state tree, and that the existing state tree nodes are deprecated, to be obsoleted over time.

- o All status "current" data nodes under the existing "state" trees are copied to the equivalent place under the "config" tree:

  - * If a node with the same name and type already exists under the equivalent path in the config tree then the nodes are merged and the description updated.

  - * If a node with the same name but different type exists under the equivalent path in the config tree, then the module authors must choose the appropriate mechanism to combine the config and state nodes in a backwards compatible way based on the data model design guidelines below. This may require the state node to be added to the config tree with a modified name. This scenario is expected to be relatively uncommon.

  - * If no node with the same name and path already exists under the config tree then the state node schema is copied verbatim into the config tree.

  - * As the state nodes are copied into the config trees, any leafrefs that reference other nodes in the state tree are adjusted to reference the equivalent path in the config tree.

  - * All status "current" nodes under the existing "state" trees are marked as "status" deprecated.

- o Augmentations are similarly handled to data nodes as described above.

C.2. Standardization of new YANG Data Models

New standards based YANG modules, or those in active development, should be designed to conform to the revised datastore architecture, following the design guidelines described below, and only need to provide combined config/state trees.

Appendix D. Implications on other Documents

The sections below describe the authors' thoughts on how various other documents may be updated to support the datastore architecture described in this document. They have been incorporated as an appendix of this document to facilitate easier review, but the expectation is that this work will be moved into another document as soon as the appropriate working group decides to take on the work.

D.1. Implications on YANG

Note: This section describes the authors' thoughts on how YANG [RFC7950] could be updated to support the datastore architecture described in this document. It has been incorporated here as a temporary measure to facilitate easier review, but the expectation is that this work will be owned and standardized via the NETCONF working group.

- o Some clarifications may be needed if this datastore model is adopted. YANG currently describes validation in terms of the <running> configuration datastore while it really happens on the <intended> configuration datastore.

D.2. Implications on YANG Library

Note: This section describes the authors' thoughts on how YANG Library [RFC7895] could be updated to support the datastore architecture described in this document. It has been incorporated here as a temporary measure to facilitate easier review, but the expectation is that this work will be owned and standardized via the NETCONF working group.

With the introduction of multiple datastores, it is important that a server can advertise to clients which modules are supported in the different datastores implemented by the server. In order to do this, we propose that the "ietf-yang-module" ([RFC7895]) is revised, with the following addition to the "module" list in the "module-list" grouping:

```
     leaf-list datastore {
       type identityref {
         base ds:datastore;
       }
       description
         "The datastores in which this module is supported.";

     }
```

D.3. Implications to YANG Guidelines

Note: This section describes the authors' thoughts on how Guidelines for Authors and Reviewers of YANG Data Model Documents [I-D.ietf-netmod-rfc6087bis] could be updated to support the datastore architecture described in this document. It has been incorporated here as a temporary measure to facilitate easier review, but the expectation is that this work will be owned and standardized via the NETCONF working group.

It is important to design data models with clear semantics that work equally well for instantiation in a configuration datastore and instantiation in the <operational> datastore.

D.3.1. Nodes with different config/state value sets

There may be some differences in the value set of some nodes that are used for both configuration and state. At this point of time, these are considered to be rare cases that can be dealt with using different nodes for the configured and state values.

D.3.2. Auto-configured or Auto-negotiated Values

Sometimes configuration leafs support special values that instruct the system to automatically configure a value. An example is an MTU that is configured to "auto" to let the system determine a suitable MTU value. Another example is Ethernet auto-negotiation of link speed. In such a situation, it is recommended to model this as two separate leafs, one config true leaf for the input to the auto-negotiation process, and one config false leaf for the output from the process.

D.4. Implications on NETCONF

Note: This section describes the authors' thoughts on how NETCONF [RFC6241] could be updated to support the datastore architecture described in this document. It has been incorporated here as a temporary measure to facilitate easier review, but the expectation is that this work will be owned and standardized via the NETCONF working group.

D.4.1. Introduction

The NETCONF protocol [RFC6241] 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.

NETCONF already has support for configuration datastores, but it does not define an operational datastore. Instead, it provides the <get> operation that returns the contents of the <running> datastore along with all config false leaves. However, this <get> operation is incompatible with the new datastore architecture defined in this document, and hence should be deprecated.

There are two possible ways that NETCONF could be extended to support the new architecture: Either as new optional capabilities extending the current version of NETCONF (v1.1, [RFC6241]), or by defining a new version of NETCONF.

Many of the required additions are common to both approaches, and are described below. A following section then describes the benefits of defining a new NETCONF version, and the additional changes that would entail.

D.4.2. Overview of additions to NETCONF

- o A new "supported datastores" capability allows a device to list all datastores it supports. Implementations can choose which datastores they expose, but MUST at least expose both the <running> and <operational> datastores. They MAY expose additional datastores, such as <intended>, <candidate>, etc.

- o A new <get-data> operation is introduced that allows the client to return the contents of a datastore. For configuration datastores, this operation returns the same data that would be returned by the existing <get-config> operation.

- o Some form of new filtering mechanism is required to allow the device to filter the data based on the YANG metadata in addition to other filters (such as the subtree filter). See also Appendix E.

- o A new "with-metadata" capability allows a device to indicate that it supports the capability of including YANG metadata annotations in the responses to <get> and <get-config> requests. This is achieved in a similar way to with-defaults [RFC6243], by introducing a <with-metadata> XML element to <get> and <get-config> requests.

* The capability would allow a device to indicate which types of metadata are supported.

* The XML element would specify which types of metadata are included in the response.

o The handling of defaults for the new configuration datastores is as described in with-defaults [RFC6243], but that does not apply for the operational state datastore that defines new semantics.

D.4.2.1. Operational State Datastore Defaults Handling

The normal semantics for the <operational> datastore are that all values that match the default specified in the schema are included in response to requests on the operational state datastore. This is equivalent to the "report-all" mode of the with-defaults handling.

The "metadata-filter" query parameter can be used to exclude nodes with origin metadata matching "default", that would exclude nodes that match the default value specified in the schema.

If the server cannot return a value for any reason (e.g., the server cannot determine the value, or the value that would be returned is outside the allowed leaf value range) then the server can choose to not return any value for a particular leaf, which MUST be interpreted by the client as the value of that leaf not being known, rather than implicitly having the default value.

D.4.3. Overview of NETCONF version 2

This section describes NETCONF version 2, by explaining the differences to NETCONF version 1.1. Where not explicitly specified, the behavior of NETCONF version 2 is the same as for NETCONF version 1.1 [RFC6241].

D.4.3.1. Benefits of defining a new NETCONF version

Defining a new version of NETCONF (as opposed to extending NETCONF version 1.1) has several benefits:

o It allows for removal of the existing <get> RPC operation, that returns content from both the running configuration datastore combined with all config false leaves.

- It could allow the existing <get-config> operation to also be removed, replaced by the more generic <get-data> that is named appropriately to also apply to the operational datastore.

- It makes it easier for clients and servers to know what reasonable common baseline functionality to expect, rather than a collection of capabilities that may not be implemented in a consistent fashion. In particular, clients will able to assume support for the <operational> datastore.

- It can gracefully coexist with NETCONF v1.1. A server could implement both versions. Existing YANG models exposing split config/state trees could be exposed via NETCONF v1.1, whereas combined config/state YANG models could be exposed via NETCONF v2, providing a viable server upgrade path.

D.4.3.2. Proposed changes for NETCONF v2

The differences between NETCONF v2 and NETCONF v1.1 can be summarized as:

- NETCONF v2 advertises a new base NETCONF capability "urn:ietf:params:netconf:base:2.0". A server may advertise older NETCONF versions as well, to allow a client to choose which version to use.

- NETCONF v2 removes support for the existing <get> operation, that is replaced by the <get-data> on the operational datastore.

- NETCONF v2 can publish a separate version of YANG library from a NETCONF v1.1 implementation running on the same device, allowing different versions of NETCONF to support a different set of YANG modules.

D.4.3.3. Possible Migration Paths

A common approach in current data models is to have two separate trees "/foo" and "/foo-state", where the former contains config true nodes, and the latter config false nodes. A data model that is designed for the revised architectural framework presented in this document will have a single tree "/foo" with a combination of config true and config false nodes.

Two different migration strategies are considered:

D.4.3.3.1. Migration Path using two instances of NETCONF

If, for backwards compatability reasons, a server intends to support both split config/state trees and the combined config/state trees proposed in this architecture, then this can be achieved by having the device support both NETCONF v1 and NETCONF v2 at the same time:

- The NETCONF v1 implementation could support existing YANG module revisions defined with split config/state trees.

- The NETCONF v2 implementation could support different YANG modules, or YANG module revisions, with combined config/state trees.

Clients can then decide on which type of models to use by expressing the appropriate version of the base NETCONF capability during capability exchange.

D.4.3.3.2. Migration Path using a single instance of NETCONF

The proposed strategy for updating existing published data models is to publish new revisions with the state trees’ nodes copied under the config tree, and for the existing state trees to have all of their nodes marked as deprecated. The expectation is that NETCONF servers would use a combination of these updated models alongside new models that only follow the new datastore architecture.

- NETCONF servers can support clients that are not aware of the revised datastore architecture, particularly if they continue to support the deprecated <get> operation:

  * For updated YANG modules they would see additional information returned via the <get> operation.

  * For new YANG modules, some of the state nodes may not be available, i.e. for any state nodes that exist under a config node that has not been configured (e.g., statistics under a system created interface).

- NETCONF servers can also support clients that are aware of the revised datastores architecture:

  * For updated YANG modules they would see additional information returned under the legacy state trees. This information can be excluded using appropriate subtree filters.

  * New YANG modules, conforming to the datastores architecture, would work exactly as expected.

D.5. Implications on RESTCONF

This section describes the authors' thoughts on how RESTCONF [RFC8040] could be updated to support the datastore architecture described in this document. It has been incorporated here as a temporary measure to facilitate easier review, but the expectation is that this work will be owned and standardized via the NETCONF working group.

D.5.1. Introduction

RESTCONF [RFC8040] defines a protocol based on HTTP for configuring data defined in YANG version 1 or 1.1, using a conceptual datastore that is compatible with a server that implements NETCONF 1.1 compliant datastores.

The combined conceptual datastore defined in RESTCONF is incompatible with the new datastore architecture defined in this document. There are two possible ways that RESTCONF could be extended to support the new architecture: Either as new optional capabilities extending the existing RESTCONF RFC, or possibly as an new version of RESTCONF.

Many of the required additions are common to both approaches, and are described below. A following section then describes the potential benefits of defining a new RESTCONF version, and the additional changes that might entail.

D.5.2. Overview of additions to RESTCONF

- o A new path {+restconf}/datastore/<datastore-name>/data/ to provide a YANG data tree for each datastore that is exposed via RESTCONF.

- o Implementations can choose which datastores they expose, but MUST at least expose both the <running> and <operational> datastores. They MAY expose the <intended> datastores as needed.

- o The same HTTP Methods supported on {+restconf}/data/ are also supported on {+restconf}/datastore/<datastore-name>/data/ but suitably constrained depending on whether the datastore can be written to by the client, or is read-only.

- o The same query parameters supported on {+restconf}/data/ are also support on {+restconf}/datastore/<datastore-name>/data/ except for the following query parameters:

- o "metadata" - is a new optional query parameter that filters the returned data based on the metadata annotation.

- o "with-metadata" - is a new optional query parameter that indicating that the metadata annotations should be included in the reply.

- o "with-defaults" is supported on all configuration datastores, but is not supported on the operational state datastore path, because it has different default handling semantics.

- o The handling of defaults (include the with-defaults query parameter) for the new configuration datastores is the same as the existing conceptual datastore, but does not apply for the operational state datastore that defines new semantics.

D.5.2.1. HTTP Methods

All configuration datastores support all HTTP Methods.

The <operational> datastore only supports the following HTTP methods: OPTIONS, HEAD, GET, and POST to invoke an RFC operation.

D.5.2.2. Query parameters

[RFC7952] specifies how a YANG data tree can be annotated with generic metadata information, that is used by this document to annotate data nodes with origin information indicating the mechanism by which the operational value came into effect.

RESTCONF could be extended with an optional generic mechanism to allow the filtering of nodes returned in a query based on metadata annotations associated with the data node.

RESTCONF could also be extended with an optional generic mechanism to choose whether metadata annotations should be included in the response, potentially filtering to a subset of annotations. E.g., only include @origin metadata annotations, and not any others that may be in use.

Both of the generic mechanisms could be controlled by a new capability. A new capability is defined to indicate whether a device supports filtering on, or annotating responses with, the origin meta data.

D.5.2.3. Operational State Datastore Defaults Handling

The normal semantics for the <operational> datastore are that all values that match the default specified in the schema are included in response to requests on the operational state datastore. This is equivalent to the "report-all" mode of the with-defaults handling.

The "metadata" query parameter can be used to exclude nodes with a origin metadata matching "default", that would exclude (only config true?) nodes that match the default value specified in the schema.

If the server cannot return a value for any reason (e.g., the server cannot determine the value, or the value that would be returned is outside the allowed leaf value range) then the server can choose to not return any value for a particular leaf, which MUST be interpreted by the client as the value of that leaf not being known, rather than implicitly having the default value.

D.5.3. Overview of a possible new RESTCONF version

This section describes a notional new RESTCONF version, by explaining the differences to RESTCONF version 1. Where not explicitly specified, the behavior of a new RESTCONF version is the same as for RESTCONF version 1 [RFC8040].

D.5.3.1. Potential benefits of defining a new RESTCONF version

Defining a new version of RESTCONF (as opposed to extending RESTCONF version 1) has several potential benefits:

- It could expose datastores, and models designed for the revised datastore architecture, in a clean and consistent way.

- It would allow the parts of RESTCONF that do not work well with the revised datastore architecture to be omitted from the new RESTCONF version.

- It would make it easier for clients and servers to know what reasonable common baseline functionality to expect, rather than a collection of capabilities that may not be implemented in a consistent fashion.

- It could gracefully coexist with RESTCONF v1. A server could implement both versions. Existing YANG models exposing split config/state trees could be exposed via RESTCONF v1, whereas combined config/state YANG models could be exposed via a new RESTCONF version, providing a viable server upgrade path.

D.5.3.2. Possible changes for a new RESTCONF version

The differences between a notional new RESTCONF version and RESTCONF version 1 (RESTCONF v1) [RFC8040] can be summarized as:

- A new RESTCONF version would define a new root resource, and a separate link relation in the /.well-known/host-meta resource.

- o A new RESTCONF version could remove support for the {+restconf}/data path supported in RESTCONF v1.

- o A new RESTCONF version could publish a separate version of YANG library from a RESTCONF v1 implementation running on the same device, allowing different versions of RESTCONF to support a different set of YANG modules.

D.5.3.3. Possible Migration Path using a new RESTCONF version

A common approach in current data models is to have two separate trees "/foo" and "/foo-state", where the former contains config true nodes, and the latter config false nodes. A data model that is designed for the revised architectural framework presented in this document will have a single tree "/foo" with a combination of config true and config false nodes.

If for backwards compatability reasons, a server intends to support both split config/state trees, and the combined config/state trees proposed in this architecture, then this could be achieved by having the device support both RESTCONF v1 and the new RESTCONF version at the same time:

- o The RESTCONF v1 implementation could support existing YANG module revisions defined with split config/state trees.

- o The implementation of the new RESTCONF version could support different YANG modules, or YANG module revisions, with combined config/state trees.

Clients can then decide on which type of models to use by choosing whether to use the RESTCONF v1 root resource or the root resource associated with the new RESTCONF version.

Appendix E. Open Issues

1. NETCONF needs to be able to filter data based on the origin metadata. Possibly this could be done as part of the <get-data> operation.

2. We need a means of inheriting @origin values, so whole hierarchies can avoid the noise of repeating parent values. Should "origin='system'" (or whatever we call it) be the default?

3. We need to discuss somewhere how remote procedure calls and notifications/actions tie into datastores. RFC 7950 shows as an example a ping action tied to an interface. Does this refer to an interface defined in a configuration datastore? Or an

interface defined in the operational state datastore? Or the applied configuration datastore? Similarly, RFC 7950 shows an example of a link-failure notification; this likely applies implicitly to the operational state datastore. The netconf-config-change notification does explicitly identify a datastore. I think we generally need to have remote procedure calls and notifications be explicit about which datastores they apply to and perhaps change the default xpath context from running plus state to the operational state datastore.

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

Rob Wilton
Cisco Systems

Email: rwilton@cisco.com