Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2015

R. Shakir
BT
A. Shaikh
M. Hines
Google

March 9, 2015

Consistent Modeling of Operational State Data in YANG
draft-openconfig-netmod-opstate-00

Abstract

This document proposes an approach for modeling configuration and operational state data in YANG that is geared toward network management systems that require capabilities beyond those typically envisioned in a NETCONF-based management system. The document presents the requirements of such systems and proposes a modeling approach to meet these requirements, along with implications and design patterns for modeling operational state in YANG.

Status of This Memo

Copyright Notice

1. Introduction

Retrieving the operational state of a network element (NE) is a critical process for a network operator, both because it determines how the network is currently running (for example, how many errors are occurring on a certain link, what is the load of that link); but also because it determines whether the intended configuration applied by a network management system is currently operational. Whilst changing of the configuration of NE may be a process which occurs relatively infrequently, the accessing of the state of the network is significantly more often - with knowledge of the real-time state of the network by external analysis and diagnostic systems being desired (implying reading of this data on the order of millisecond timescales).

It is desirable to model configuration and operational state within a single schema. This allows a network operator or management system to retrieve information as to the intended state of the network system (the configuration), and easily relate to this to the actual running state. There are numerous reasons that the intended state may not be reflected by the running config:

- o Protocol negotiations may be required for multiple NEs to agree on a certain running value - for example, the HOLDTIME of a BGP session is chosen by taking the minimum value of the locally configured and advertised value received from the remote speaker. The operational value of the HOLDTIME may therefore be lower than the configured value on the local system.

- o Where the application of a change is asynchronous - due to system operations, or a pre-requisite for another event to occur before the configuration value is applied (e.g., a protocol session restart) - then the intended configuration value may not determine whether the configuration has been committed to the running configuration; or whether the pre-requisite event has occurred.

Based on such differences between intended and running state, the operation of checking one state and then subsequently applying a change is very common. For example, checking the current IGP metric of a certain link, and if it is not reflective of the desired value, subsequently applying a change. Rather than viewing configuration and operational state separately, having both types of values in a common location within the same data schema is advantageous. In this way, no complex mapping between the path where the value is read, and the path by which it is configured is required.

The majority of existing designs of the layout and presentation of a YANG [RFC6020] model considers only the semantics of the NETCONF protocol - however, it is advantageous that any data model (expressed in YANG) should be suitable for use with multiple protocols. Such protocols may be alternatives to NETCONF - e.g., RESTCONF - but also may be specifically engineered for accessing particular operational data (e.g., streamed data from a network element, rather than 'SNMP-like' polled mechanisms).

Based on the inherent link between the configuration and state data for a NE, and the importance of state for a network operator, YANG's focus solely on configuration data is suboptimal[RFC6244]. We therefore consider that there is a requirement to consider (and define common approaches for) the definition of state and operational data within a YANG model. Such considerations should be cognisant of the protocols used to interact with the data schema.

2. Operational requirements

Our proposal is motivated by a number of operational requirements as described below.

2.1. Intended configuration as part of operational state

The definition of operational state in [RFC6244] includes read-only transient data that is the result of system operation or protocol interactions, and data that is typically thought of as counters or statistics. In many operational use cases it is also important to distinguish between the intended value of a configuration variable and its actual configured state, analogous to candidate and running configuration, respectively, in NETCONF datastores. In non-transactional or asynchronous environments, for example, these may be different and it is important to know when they are different or when they have converged (see requirement #2). For this reason, we consider the intended configuration as an additional important element of the operational state. This is not considered in [RFC6244].

2.2. Support for both transactional, synchronous management systems as well as distributed, asynchronous management systems

In a synchronous system, configuration changes are transactional and committed as an atomic unit. This implies that the management system knows the success or failure of the configuration change based on the return value, and hence knows that the intended configuration matches what is on the system. In particular, the value of any configuration variable should always reflect the (intended) configured value. Synchronous operation is generally associated with a NETCONF-based system that provides transactional semantics for all changes.

In an asynchronous system, configuration changes to the system may not be reflected immediately, even though the change operation returns success. Rather, the change is verified by observing the state of the system, for example based on notifications, or continuously streamed values of the state. In this case, the value of a configuration variable may not reflect the intended configured value at a given point in time.

The asynchronous use case is important because synchronous operation may not always be possible. For example, in a large scale environment, the management system may not need to wait for all changes to complete if it is acceptable to proceed while some configuration values are being updated. In addition, not all devices may support transactional changes, making asynchronous operation a requirement. Moreover, using observed state to infer the configured value allows the management system to learn the time taken to complete various configuration changes.

2.3. Separation of configuration and operational state data; ability to retrieve them independently

These requirements are also mentioned in [RFC3535]:

- o It is necessary to make a clear distinction between configuration data, data that describes operational state and statistics.

- o It is required to be able to fetch separately configuration data, operational state data, and statistics from devices, and to be able to compare these between devices.

2.4. Ability to retrieve operational state corresponding only to derived values, statistics, etc.

When the management system operates in synchronous mode, it should be able to retrieve only the operational state corresponding to the system determined values, such as negotiated values, protocol determined values, or statistics and counters. Since in synchronous mode the intended and actual configuration values are identical, sending the intended configuration state is redundant.

2.5. Consistent schema locations for configuration and corresponding operational state data

This requirement implies that a common convention is used throughout the schema to locate configuration and state data so that the management system can infer how to access one or the other without needing significant external context. When considering intended configuration as part of operational state (as discussed in Section 2.1), it is similarly required that the intended value vs. actual value for a particular configuration variable should be possible to locate with minimal, if any, mapping information.

This requirement becomes more evident when considering the composition of individual data models into a higher-level model for a complete device (e.g., /device[name=devXY]/protocols/routing/...) or even higher layer models maintained by network operators (e.g., /operatorX/global/continent[name=eur]/pop[name=paris]/device[name=devXY]/...). If each model has it's own way to separate configuration and state data, then this information must be known at potentially every subtree of the composed model.

3. Implications on modeling operational state

The requirements in Section 2 give rise to a number of new considerations for modeling operational state. Some of the key implications are summarized below.

3.1. Inclusion of intended configuration as part of operational state

This implies that a copy of the configurable (i.e., writable) values should be included as read-only variables in containers for operational state, in addition to the variables that are traditionally thought of as state variables (counters, negotiated values, etc.).

3.2. Corresponding leaves for configuration and state

Any configuration leaf should have a corresponding state leaf. The opposite is clearly not true -- some parts of the model may only have derived state variables, for example the contents of a routing table that is populated by a dynamic routing protocols like BGP or IS-IS.

3.3. Retrieval of only the derived, or NE-generated part of the operational state

YANG and NETCONF do not currently differentiate between state that is derived by the NE, state representing statistics, and state representing intended configuration -- all state is simply marked as 'config false' or read-only. To retrieve only the state that is not part of intended configuration, we require a new way to tag such data. This is proposed in this document as a YANG extension. Alternatively, as described in [RFC6244], a new NETCONF datastore for operational state that is just for NE- generated state could also be used to allow <get> (or similar) operations to specify just that part of the state.

3.4. Consistency and predictability in the paths where corresponding state and configuration data may be retrieved

To avoid arbitrary placement of state and configuration data containers, the most consistent options would be at the root of the model (as done in [YANG-IF]) or at the leaves, i.e., at the start or end of the paths. When operators compose models into a higher level model, the root of the model is no longer well-defined, and hence neither is the start of the path. For these reasons, we propose placing configuration and state separation at leaves of the model.

3.5. Reuse of existing NETCONF conventions where applicable

Though not a specific requirement, models for operational state should take advantage of existing protocol mechanisms where possible, e.g., to retrieve configuration and state data.

4. Proposed operational state structure

Below we show an example model structure that meets the requirements described above for all four types of data we are considering:

- configuration (writable) data
- operational state data on the NE that is derived, negotiated, set by a protocol, etc.
- operational state data for counters or statistics
- operational state data representing intended configuration

4.1. Example model structure

The example below shows a partial model (in ascii tree format) for managing Ethernet aggregate interfaces (leveraging data definitions from [RFC7223]):

```
+--rw interfaces
    +--rw interface* [name]
       +--rw name       -> ../config/name
       +--rw config
       |    ...
       +--ro state
       |  | ...
       |  +--ro counters
       |     +--ro discontinuity-time    yang:date-and-time
       |     +--ro in-octets?            yang:counter64
       |     +--ro in-unicast-pkts?      yang:counter64
       |     +--ro in-broadcast-pkts?    yang:counter64
       |     +--ro in-multicast-pkts?    yang:counter64
       |     +--ro in-discards?          yang:counter64
       |     +--ro in-errors?            yang:counter64
       |     +--ro in-unknown-protos?    yang:counter64
       |     +--ro out-octets?           yang:counter64
       |     +--ro out-unicast-pkts?     yang:counter64
       |     +--ro out-broadcast-pkts?   yang:counter64
       |     +--ro out-multicast-pkts?   yang:counter64
       |     +--ro out-discards?         yang:counter64
       |     +--ro out-errors?           yang:counter64
       +--rw aggregation!
          +--rw config
          |  +--rw lag-type?    aggregation-type
          |  +--rw min-links?   uint16
          +--ro state
          |  +--ro lag-type?    aggregation-type
          |  +--ro min-links?   uint16
          |  +--ro members*     ocif:interface-ref
          +--rw lacp!
             +--rw config
             |  +--rw interval?   lacp-period-type
             +--rw members* [interface]
             |  +--rw interface    ocif:interface-ref
             |  +--ro state
             |     +--ro activity?          lacp-activity-type
             |     +--ro timeout?           lacp-timeout-type
             |     +--ro synchronization?   lacp-synch-type
             |     +--ro aggregatable?      boolean
             |     +--ro collecting?        boolean
             |     +--ro distributing?      boolean
             +--ro state
                +--ro interval?   lacp-period-type
```

In this model, the path to the configurable (rw) items at the aggregate interface level is:

/interfaces/interface[name=ifName]/aggregation/config/...

The corresponding operational state is located at:

/interfaces/interface[name=ifName]/aggregation/state/...

This container holds a read-only copy of the intended configuration variables (lag-type and min-links), as well as a generated list of member interfaces (the members leaf-list) for the aggregate that is active when the lag-type indicates a statically configured aggregate. Note that although the paths to config and state containers are symmetric, the state container contains additional derived variables.

The model has an additional hierarchy level for aggregate interfaces that are maintained using LACP. For these, the configuration path is:

/interfaces/interface[name=ifName]/aggregation/lacp/config/...

with the corresponding state container (in this case with only the state corresponding to the intended configuration) at:

/interfaces/interface[name=ifName]/aggregation/lacp/state/...

There is an additional list of members for LACP-managed aggregates with only a state container:

/interfaces/interface[name=ifName]/aggregation/lacp/
members[name=ifName]/state/...

Note that it is not required that both a state and a config container be present at every leaf. It may be convenient to include an empty config container to make it more explicit to the management system that there are no configuration variables at this location in the data tree.

Finally, we can see that the generic interface object also has config and state containers (these are abbreviated for clarity). The state container has a subcontainer for operational state corresponding to counters and statistics that are valid for any interface type:

/interfaces/interface[name=ifName]/state/counters/...

5. Impact on model authoring

One drawback of structuring operational and configuration data in this way is the added complexity in authoring the models, relative to the way some models are currently built with state and config split at the root of the individual model (e.g., in [RFC7223], [RFC7317], and [IETF-RTG]). Moving the config and state containers to each leaf adds a one-time modeling effort, which is somewhat dependent on the model structure itself (how many layers of container hierarchy, number of lists, etc.) However, we feel this effort is justified by the resulting simplicity with which management systems can access and correlate state and configuration data.

5.1. Modeling design patterns

We propose some specific YANG modeling design patterns that may be useful for building models following these conventions.

5.1.1. Basic structure

Since leaves that are created under the 'config' container are duplicated under the 'state' container, it is recommended that the following conventions are used to ensure that the schema remain as simple as possible:

o A grouping for the 'config' data items is created - with a specific naming convention to indicate that such variables are configurable, such as a suffix like '-config' or '_config'. For example, the OpenConfig BGP model [OC-BGP] adopts the convention of appending "_config" to the name of the container.

o A grouping for the 'state' data items is created, with a similar naming convention as above, i.e., with a suffix such as '-state' or '_state'. The BGP model uses "_state".

o A structure grouping is created that instantiates both the 'config' and 'state' containers. The 'config' container should include the "-config" grouping, whilst the state container has both the "-config" and "-state" groupings, along with the 'config false' statement.

A simple example in YANG is shown in Appendix B.

5.1.2. Handling lists

In YANG 1.0, lists have requirements that complicate the creation of the parallel configuration and state data structures. First, keys must be children of the list; they cannot be further down the data hierarchy within a subsequent container. For example, the 'interface' list cannot be keyed by /interfaces/interface/config/ name. Second YANG requires that the list key is part of the configuration or state data in each list member.

We consider two possible approaches for lists:

1. list keys appear only at the top level of the list, i.e., not duplicated under the 'config' or 'state' containers within the list

2. the data represented by the list key appears in the config and state containers, and a key with type leafref is used in the top level of the list pointing to the corresponding data node in the config (or state) container.

Option 1 has the advantage of not duplicating data, but treats the data item (or items) that are keys as special cases, i.e., not included in the config or state containers. Option 2 is appealing in that configurable data always appears in the config container, but requires an arguably unnecessary key pointing to the data from the top level of the list.

Appendix C shows a simple example of both options.

5.1.3. Selective use of state data from common groupings

In a number of cases, it is desirable that the same grouping be used within different places in a model - but state information is only relevant in one of these paths. For example, considering BGP, peer configuration is relevant to both a "neighbor" (i.e., an individual BGP peer), and also to a peer-group (a set of peers). Counters relating to the number of received prefixes, or queued messages, are relevant only within the 'state' container of the peer (rather than the peer-group). In this case, use of the 'augment' statement to add specific leaves to only one area of the tree is recommended, since it allows a common container to be utilized otherwise.

5.1.4. Non-corresponding configuration and state data

There are some instances where only an operational state container is relevant without a corresponding configuration data container. For example, the list of currently active member interfaces in a LACP-managed LAG is typically reported by the system as operational state that is governed by the LACP protocol. Such data is not directly configured. Similarly, counters and statistics do not have corresponding configuration. In these cases, we can either omit the config container from such leaves, or provide an empty container as described earlier. With both options, the management system is able to infer that such data is not configurable.

6. YANG language considerations

In adopting the approach described in this document for modeling operational state data in YANG, we encounter several language limitations that are described below. We disucss some initial thoughts on possible changes to the language to more easily enable the proposed model for operational state modeling.

6.1. Distinguishing derived operational state data and intended configuration

As mentioned in Section 2, we require a way to separately query operational state that is not part of intended configuration (e.g., protocol-determined data, counters, etc.). YANG and NETCONF do not distinguish types of operational state data, however. To overcome this, we currently use a YANG language extension to mark such data as 'operational: true'. Ideally, this could be generalized beyond the current 'config: true / false' to something like "operational: false", "operational: intent", and "operational:true".

6.2. YANG lists as maps

YANG has two list constructs, the 'leaf-list' which is similar to a list of scalars in other programming languages, and the 'list' which allows a keyed list of complex structures, where the key is also part of the data values. As described in Section [impact], the current requirements on YANG list keys require either duplication of data, or treating some data (i.e., those that comprise list keys) as a special case. One solution is to generalize lists to be more like map data structures, where each list member has a key that is not required to part of the configuration or state data. This allows list keys to be arbitrarily defined by the user if desired, or based on values of data nodes. In the latter case, the specification of which data nodes are used in constructing the list key could be indicated in the meta-data associated with the key.

6.3. Configuration and state data hierarchy

YANG does not allow read-write configuration data to be child nodes of read-only operational state data. This requires the definition of separate state and config containers as described above. However, it may be desirable to simplify the schema by 'flattening', e.g., having the operational state as the root of the data tree, with only config containers needed to specify the variables that are writable (in general, the configuration data is much smaller than operational state data). Naming the containers explicitly according the config / state convention makes the intent of the data clear, and should allow relaxing of the current YANG restrictions. That is, a read-write config container makes explicit the nature of the enclosed data even if the parent data nodes are read-only. This of course requires that all data in a config container are in fact configurable -- this is one of the motivations of pushing such containers as far down in the schema hierarchy as possible.

7. Security Considerations

This document addresses the structure of configuration and operational state data, both of which should be considered sensitive from a security standpoint. Any data models that follows the proposed structuring must be carefully carefully evaluated to determine its security risks. In general, access to both configuration (write) and operational state (read) data must be carefully controlled through appropriate access control and authorization mechanisms.

8. References

8.1. Normative references

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

[RFC6244] Shafer, P., "An Architecture for Network Management Using NETCONF and YANG", RFC 6244, June 2011.

[RFC3535] Schoenwaelder, J., "Overview of the 2002 IAB Network Management Workshop", RFC 3535, May 2003.

[RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, May 2014.

[RFC7317] Bierman, A. and M. Bjorklund, "A YANG Data Model for System Management", RFC 7317, August 2014.

8.2. Informative references

[IETF-RTG] Lhotka, L., "A YANG Data Model for Routing Management", draft-ietf-netmod-routing-cfg-16 (work in progress), October 2014.

[OC-BGP] Shaikh, A., D’Souza, K., Bansal, D., and R. Shakir, "BGP Configuration Model for Service Provider Networks", draft-shaikh-idr-bgp-model-01 (work in progress), March 2015.

Appendix A. Acknowledgements

The authors are grateful for valuable input to this document from: Martin Bjorklund, Paul Borman, Chris Chase, Feihong Chen, Josh George, Carl Moberg, Jason Sterne, and Jim Uttaro.

Appendix B. Example YANG base structure

Below we show an example of the basic YANG building block for organizing configuration and operational state data as described in Section 4

```
grouping example-config {
    description "configuration data for example container";

    leaf conf-1 {
      type empty;
    }

    leaf conf-2 {
      type string;
    }
  }

  grouping example-state {
    description
      "operational state data (derived, counters, etc.) for example
      container";

    leaf state-1 {
      type boolean;
    }

    leaf state-2 {
      type string;
    }

    container counters {
      description
        "operational state counters for example container";

      leaf counter-1 {
        type uint32;
      }

      leaf counter-2 {
        type uint64;
      }
```

```
       }
     }

     grouping example-structure {
       description
         "top level grouping for the example container -- this is used
         to put the config and state subtrees in the appropriate
         location";

       container example {
         description
           "top-level container for the example data";

         container config {

           uses example-config;

         }

         container state {

           config false;
           uses example-config;
           uses example-state;
         }
       }
     }

     uses example-structure;
```

The corresponding YANG data tree is:

```
      +--rw example
         +--rw config
         |  +--rw conf-1?   empty
         |  +--rw conf-2?   string
         +--ro state
            +--ro conf-1?     empty
            +--ro conf-2?     string
            +--ro state-1?    boolean
            +--ro state-2?    string
            +--ro counters
               +--ro counter-1?   uint32
               +--ro counter-2?   uint64
```

Appendix C. Example YANG list structure

As described in Section 5.1.2, there are two options we consider for building lists according to the proposed structure. Both are shown in the example YANG snippet below. The groupings defined above in Appendix B are reused here.

```
grouping example-no-conf2-config {
    description
    "configuration data for example container but without the conf-2
    data leaf which is used as a list key";

    leaf conf-1 {
      type empty;
    }

  }

  grouping example-structure {
    description
      "top level grouping for the example container -- this is used
      to put the config and state subtrees in the appropriate
      location";

    list example {

      key conf-2;
      description
        "top-level list for the example data";

      leaf conf-2 {
        type leafref {
          path "../config/conf-2";
        }
      }

      container config {

        uses example-config;

      }

      container state {

        config false;
        uses example-config;
        uses example-state;
      }
```

```
       }

       list example2 {

         key conf-2;
         description
           "top-level list for the example data";

         leaf conf-2 {
           type string;
         }

         container config {

           uses example-no-conf2-config;

         }

         container state {

           config false;
           uses example-no-conf2-config;
           uses example-state;
         }
       }
     }

     uses example-structure;
```

The corresponding YANG data tree is shown below for both styles of lists.

```
   +--rw example* [conf-2]
      |  +--rw conf-2    -> ../config/conf-2
      |  +--rw config
      |  |  +--rw conf-1?   empty
      |  |  +--rw conf-2?   string
      |  +--ro state
      |     +--ro conf-1?     empty
      |     +--ro conf-2?     string
      |     +--ro state-1?    boolean
      |     +--ro state-2?    string
      |     +--ro counters
      |        +--ro counter-1?   uint32
      |        +--ro counter-2?   uint64
      +--rw example2* [conf-2]
         +--rw conf-2    string
         +--rw config
         |  +--rw conf-1?   empty
         +--ro state
            +--ro conf-1?     empty
            +--ro state-1?    boolean
            +--ro state-2?    string
            +--ro counters
               +--ro counter-1?   uint32
               +--ro counter-2?   uint64
```

Authors' Addresses

Rob Shakir
BT
pp. C3L, BT Centre
81, Newgate Street
London EC1A 7AJ
UK

Email: rob.shakir@bt.com
URI: http://www.bt.com/

Anees Shaikh
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
US

Email: aashaikh@google.com

Marcus Hines
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
US

Email: hines@google.com