Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 15, 2016

A. Clemm
A. Gonzalez Prieto
E. Voit
Cisco Systems
October 13, 2015

Subscribing to YANG datastore push updates
draft-clemm-netconf-yang-push-02.txt

Abstract

This document defines a subscription and push mechanism for YANG datastores. This mechanism allows client applications to request updates from a YANG datastore, which are then pushed by the server to the client per a subscription policy, without requiring additional client requests.

Table of Contents

1. Introduction

YANG [RFC6020] was originally designed for the Netconf protocol [RFC6241], which originally put most emphasis on configuration. However, YANG is not restricted to configuration data. YANG datastores, i.e. datastores that contain data modeled according using YANG, can contain configuration as well as operational data. It is therefore reasonable to expect that data in YANG datastores will increasingly be used to support applications that are not focused on managing configurations but that are, for example, related to service assurance.

Service assurance applications typically involve monitoring operational state of networks and devices; of particular interest are changes that this data undergoes over time. Likewise, there are applications in which data and objects from one datastore need to be made available both to applications in other systems and to remote datastores [I-D.voit-netmod-peer-mount-requirements] [I-D.clemm-netmod-mount]. This requires mechanisms that allow remote systems to become quickly aware of any updates to allow to validate and maintain cross-network integrity and consistency.

Traditional approaches to remote network state visibility rely heavily on polling. With polling, data is periodically explicitly retrieved by a client from a server to stay up-to-date.

There are various issues associated with polling-based management:

- It introduces additional load on network, devices, and applications. Each polling cycle requires a separate yet arguably redundant request that results in an interrupt, requires parsing, consumes bandwidth.

- It lacks robustness. Polling cycles may be missed, requests may be delayed or get lost, often particularly in cases when the network is under stress and hence exactly when the need for the data is the greatest.

- Data may be difficult to calibrate and compare. Polling requests may undergo slight fluctuations, resulting in intervals of different lengths which makes data hard to compare. Likewise, pollers may have difficulty issuing requests that reach all devices at the same time, resulting in offset polling intervals which again make data hard to compare.

A more effective alternative is when an application can request to be automatically updated as necessary of current content of the datastore (such as a subtree, or data in a subtree that meets a certain filter condition), and in which the server that maintains the datastore subsequently pushes those updates. However, such a solution does not currently exist.

The need to perform polling-based management is typically considered an important shortcoming of management applications that rely on MIBs polled using SNMP [RFC1157]. However, without a provision to support a push-based alternative, there is no reason to believe that management applications that operate on YANG datastores using protocols such as NETCONF or Restconf [I-D.ietf-netconf-restconf] will be any more effective, as they would follow the same request/response pattern.

While YANG allows the definition of notifications, such notifications are generally intended to indicate the occurrence of certain well-specified event conditions, such as a the onset of an alarm condition or the occurrence of an error. A capability to subscribe to and deliver event notifications has been defined in [RFC5277]. In addition, configuration change notifications have been defined in [RFC6470]. These change notifications pertain only to configuration information, not to operational state, and convey the root of the subtree to which changes were applied along with the edits, but not the modified data nodes and their values.

Accordingly, there is a need for a service that allows client applications to subscribe to updates of a YANG datastore and that allows the server to push those updates. The requirements for such a service are documented in [I-D.i2rs-pub-sub-requirements]. This document proposes a solution that features the following capabilities:

- A mechanism that allows clients to subscribe to automatic datastore updates, and the means to manage those subscription. The subscription allows clients to specify which data they are interested in, and to provide optional filters with criteria that data must meet for updates to be sent. Furthermore, subscription can specify a policy that directs when updates are provided. For

example, a client may request to be updated periodically in certain intervals, or whenever data changes occur.

- The ability for a server to push back on requested subscription parameters. Because not every server may support every requested interval for every piece of data, it is necessary for a server to be able to indicate whether or not it is capable of supporting a requested subscription, and possibly allow to negotiate subscription parameters.

- A mechanism to communicate the updates themselves. For this, the proposal leverages and extends existing YANG/Netconf/Restconf mechanisms, defining special notifications that carry updates.

This document specifies a YANG data model to manage subscriptions to data in YANG datastores, and to configure associated filters and data streams. It defines extensions to RPCs defined in [RFC5277] that allow to extend notification subscriptions to subscriptions for datastore updates. It also defines a notification that can be used to carry data updates and thus serve as push mechanism.

2. Definitions and Acronyms

Data node: An instance of management information in a YANG datastore.

Data record: A record containing a set of one or more data node instances and their associated values.

Datastore: A conceptual store of instantiated management information, with individual data items represented by data nodes which are arranged in hierarchical manner.

Datastream: A continuous stream of data records, each including a set of updates, i.e. data node instances and their associated values.

Data subtree: An instantiated data node and the data nodes that are hierarchically contained within it.

NACM: NETCONF Access Control Model

NETCONF: Network Configuration Protocol

Push-update stream: A conceptual data stream of a datastore that streams the entire datastore contents continuously and perpetually.

RPC: Remote Procedure Call

SNMP: Simple Network Management Protocol

Subscription: A contract between a client ("subscriber") and a server ("publisher"), stipulating which information the client wishes to receive from the server (and which information the server has to provide to the client) without the need for further solicitation.

Subscription filter: A filter that contains evaluation criteria which are evaluated against YANG objects of a subscription. An update is only published if the object meets the specified filter criteria.

Subscription policy: A policy that specifies under what circumstances to push an update, e.g. whether updates are to be provided periodically or only whenever changes occur.

Update: A data item containing the current value of a data node.

Update trigger: A trigger, as specified by a subscription policy, that causes an update to be sent, respectively a data record to be generated. An example of a trigger is a change trigger, invoked when the value of a data node changes or a data node is created or deleted, or a time trigger, invoked after the laps of a periodic time interval.

URI: Uniform Resource Identifier

YANG: A data definition language for NETCONF

Yang-push: The subscription and push mechanism for YANG datastores that is specified in this document.

3. Solution Overview

This document specifies a solution that allows clients to subscribe to information updates in a YANG datastore, which are subsequently pushed from the server to the client.

Subscriptions are initiated by clients. Servers respond to a subscription request explicitly positively or negatively. Negative responses include information about why the subscription was not accepted, in order to facilitate converging on an acceptable set of subscription parameters. Once a subscription has been established, datastore push updates are pushed from the server to the subscribing client until the subscription ends.

Accordingly, the solution encompasses several components:

- The subscription model for configuration and management of the subscriptions, with a set of associated services.

- The ability to provide hints for acceptable subscription parameters, in cases where a subscription desired by a client cannot currently be served.

- The stream of datastore push updates.

In addition, there are a number of additional considerations, such as the tie-in of the mechanisms with security mechanisms. Each of those aspects will be discussed in the following subsections.

3.1. Subscription Model

Yang-push subscriptions are defined using a data model that is itself defined in YANG. This model is based on the subscriptions defined in [RFC5277], which are also reused in Restconf. The model is extended with several parameters, including a subscription type and a subscription ID.

The subscription model assumes the presence of a conceptual perpetual datastream "push-update" of continuous datastore updates that can be subscribed to, although other datastreams may be supported as well. A subscription refers to a datastream and specifies filters that are to be applied to, it for example, to provide only those subsets of the information that match a filter criteria. In addition, a subscription specifies a set of subscription parameters that define the trigger when data records should be sent, for example at periodic intervals or whenever underlying data items change.

The complete set of subscription parameters is as follows:

- The stream being subscribed to. The subscription model assumes the presence of perpetual and continuous streams of updates. The stream "push-update" is always available and covers the entire set of YANG data in the server, but a system may provide other streams to choose from.

- The datastore to target. By default, the datastore will always be "running". However, it is conceivable that implementations want to also support subscriptions to updates to other datastores.

- An encoding for the data updates. By default, updates are encoded using XML, but JSON can be requested as an option and other encodings may be supported in the future.

- An optional start time for the subscription. If the specified start time is in the past, the subscription goes into effect immediately. The start time also serves as anchor time for

periodic subscriptions, from which intervals at which to send updates are calculated (see also below).

- An optional stop time for the subscription. Once the stop time is reached, the subscription is automatically terminated.

- A subscription policy definition regarding the update trigger when to send new updates. The trigger can be periodic or based on change.

  * For periodic subscriptions, the trigger is defined by a parameter that defines the interval with which updates are to be pushed. The start time of the subscription serves as anchor time, defining one specific point in time at which an update needs to be sent. Update intervals always fall on the points in time that are a multiple of a period after the start time.

  * For on-change subscriptions, the trigger occurs whenever a change in the subscribed information is detected. On-change subscriptions have more complex semantics that can be guided by additional parameters. Please refer also to Section 3.3.

    + One parameter is needed to specify the dampening period, i.e. the interval that must pass before a successive update for the same data node is sent. The first time a change is detected, the update is sent immediately. If a subsequent change is detected, another update is only sent once the dampening period has passed, containing the value of the data node that is then valid.

    + Another parameter allows to restrict the types of changes for which updates are sent (changes to object values, object creation or deletion events). It is conceivable to augment the data model with additional parameters in the future to specify even more refined policies, such as parameters that specify the magnitude of a change that must occur before an update is triggered.

    + A third parameter specifies whether or not a complete update with all the subscribed data should be sent at the beginning of a subscription.

- Optionally, a filter, or set of filters, describing the subset of data items in the stream's data records that are of interest to the subscriber. The server should only send to the subscriber the data items that match the filter(s), when present. The absence of a filter indicates that all data items from the stream are of interest to the subscriber and all data records must be sent in

their entirety to the subscriber. Two types of filters are support: subtree filter, with the same semantics as defined in [RFC 6241], and XPath filters. Additional filter types can be added through augmentations. Filters can be specified "inline" as part of the subscription, or can be configured separately and referenced by a subscription, in order to facilitate reuse of complex filters.

The subscription data model is specified as part of the YANG data model described later in this specification. Specifically, the subscription parameters are defined in the "subscription-info" grouping. It is conceivable that additional subscription parameters might be added in the future. This can be accomplished through augmentation of the subscription data model.

3.2. Negotiation of Subscription Policies

A subscription rejection can be caused by the inability of the server to provide a stream with the requested semantics. For example, a server may not be able to support "on-change" updates for operational data, or only support them for a limited set of data nodes. Likewise, a server may not be able to support a requested updated frequency, or a requested encoding.

Yang-push supports a simple negotiation between clients and servers for subscription parameters. The negotiation is limited to a single pair of subscription request and response. For negative responses, the server SHOULD include in the returned error what subscription parameters would have been accepted for the request. The returned acceptable parameters constitute suggestions that, when followed, increase the likelihood of success for subsequent requests. However, they are no guarantee that subsequent requests for this client or others will in fact be accepted.

In case a subscriber requests an encoding other than XML, and this encoding is not supported by the server, the server simply indicates in the response that the encoding is not supported.

3.3. On-Change Considerations

On-change subscriptions allow clients to subscribe to updates whenever changes to objects occur. As such, on-change subscriptions are of particular interest for data that changes relatively infrequently, yet that require applications to be notified with minimal delay when changes do occur.

On-change subscriptions tend to be more difficult to implement than periodic subscriptions. Specifically, on-change subscriptions may

involve a notion of state to see if a change occurred between past and current state, or the ability to tap into changes as they occur in the underlying system. Accordingly, on-change subscriptions may not be supported by all implementations or for every object.

When an on-change subscription is requested for a datastream with a given subtree filter, where not all objects support on-change update triggers, the subscription request MUST be rejected. As a result, on-change subscription requests will tend to be directed at very specific, targeted subtrees with only few objects.

Any updates for an on-change subscription will include only objects for which a change was detected. To avoid flooding clients with repeated updates for fast-changing objects, or objects with oscillating values, an on-change subscription allows for the definition of a dampening period. Once an update for a given object is sent, no other updates for this particular object are sent until the end of the dampening period. In addition, updates include information about objects that were deleted and ones that were newly created.

On-change subscriptions can be refined to let users subscribe only to certain types of changes, for example, only to object creations and deletions, but not to modifications of object values.

Additional refinements are conceivable. For example, in order to avoid sending updates on objects whose values undergo only a negligible change, additional parameters might be added to an on-change subscription specifying a policy that states how large or "significant" a change has to be before an update is sent. A simple policy is a "delta-policy" that states, for integer-valued data nodes, the minimum difference between the current value and the value that was last reported that triggers an update. Also more sophisticated policies are conceivable, such as policies specified in percentage terms or policies that take into account the rate of change. While not specified as part of this draft, such policies can be accommodated by augmenting the subscription data model accordingly.

3.4. Data Encodings

Subscribed data is encoded in either XML or JSON format. A server MUST support XML encoding and MAY support JSON encoding.

It is conceivable that additional encodings may be supported as options in the future. This can be accomplished by augmenting the subscription data model with additional identity statements used to refer to requested encodings.

3.4.1. Periodic Subscriptions

In a periodic subscription, the data included as part of an update corresponds to data that could have been simply retrieved using a get operation and is encoded in the same way. XML encoding rules for data nodes are defined in [RFC6020]. JSON encoding rules are defined in [I-D.ietf-netmod-yang-json]. This encoding is valid JSON, but also has special encoding rules to identify module namespaces and provide consistent type processing of YANG data.

3.4.2. On-Change Subscriptions

In an on-change subscription, updates need to allow to differentiate between data nodes that were newly created since the last update, data nodes that were deleted, and data nodes whose value changed.

XML encoding rules correspond to how data would be encoded in input to Netconf edit-config operations as specified in [RFC6241] section 7.2, adding "operation" attributes to elements in the data subtree. Specifically, the following values will be utilized:

- create: The data identified by the element has been added since the last update.
- delete: The data identified by the element has been deleted since the last update.
- merge: The data identified by the element has been changed since the last update.
- replace: The data identified by the element has been replaced with the update contents since the last update.

The remove value will not be utilized.

Contrary to edit-config operations, the data is sent from the server to the client, not from the client to the server, and will not be restricted to configuration data.

JSON encoding rules are roughly analogous to how data would be encoded in input to a YANG-patch operation, as specified in [I-D.ietf-netconf-yang-patch] section 2.2. However, no edit-ids will be needed. Specifically, changes will be grouped under respective "operation" containers for creations, deletions, and modifications.

3.5. Subscription Filters

Subscriptions can specify filters for subscribed data. The following filters are supported:

- subtree-filter: A subtree filter specifies a subtree that the subscription refers to. When specified, updates will only concern data nodes from this subtree. Syntax and semantics correspond to that specified for [RFC6241] section 6.

- xpath-filter: An XPath filter specifies an XPath expression applied to the data in an update, assuming XML-encoded data.

If multiple subscription filters are specified, all of them are applied. In other words, it is possible to (for example) apply an XPath filter on top of a subtree filter.

It is conceivable for implementations to support other filters. For example, an on-change filter might specify that changes in values should be sent only when the magnitude of the change since previous updates exceeds a certain threshold. It is possible to augment the subscription data model with additional filter types.

3.6. Push Data Stream and Transport Mapping

Pushing data based on a subscription could be considered analogous to a response to a data retrieval request, e.g. a "get" request. However, contrary to such a request, multiple responses to the same request may get sent over a longer period of time.

A more suitable mechanism is therefore that of a notification. Contrary to notifications associated with alarms and unexpected event occurrences, push updates are solicited, i.e. tied to a particular subscription which triggered the notification. (An alternative conceptual model would consider a subscription an "opt-in" filter on a continuous stream of updates.)

The notification contains several parameters:

- A subscription correlator, referencing the name of the subscription on whose behalf the notification is sent.

- A data node that contains a representation of the datastore subtree containing the updates. The subtree is filtered per access control rules to contain only data that the subscriber is authorized to see. Also, depending on the subscription type, i.e., specifically for on-change subscriptions, the subtree contains only the data nodes that contain actual changes. (This

can be simply a node of type string or, for XML-based encoding, anyxml.)

Notifications are sent using <notification> elements as defined in [RFC5277]. Alternative transports are conceivable but outside the scope of this specification.

The solution specified in this document uses notifications to communicate datastore updates. The contents of the notification includes a set of explicitly defined data nodes. For this purpose, two new generic notifications are introduced, "push-update" and "push-change-update". Both notifications are used to define how to carry data records with updates of datastore contents as specified by a subscription.

Push-update notification defines updates for a periodic subscription, as well as for the initial update of an on-change subscription used to synchronize the receiver at the start of a new subscription. The update record contains a data snippet that contains an instantiated subtree with the subscribed contents. The content of the update record is equivalent to the contents that would be obtained had the same data been explicitly retrieved using e.g. a Netconf "get"-operation, with the same filters applied.

The contents of the notification conceptually represents the union of all data nodes in the yang modules supported by the server. However, in a YANG data model, it is not practical to model the precise data contained in the updates as part of the notification. This is because the specific data nodes supported depend on the implementing system and may even vary dynamically. Therefore, to capture this data, a single parameter that can represent any datastore contents is used, not parameters that represent data nodes one at a time.

Push-change-update notification defines updates for on-change subscriptions. The update record here contains a data snippet that indicates the changes that data nodes have undergone, i.e. that indicates which data nodes have been created, deleted, or had changes to their values. The format follows the same format that operations that apply changes to a data tree would apply, indicating the creates, deletes, and modifications of data nodes.

The following is an example of push notification. It contains an update for subscription my-sub, including a subtree with root foo that contains a leaf, bar:

```
<?xml version="1.0" encoding="UTF-8"?>
   <notification
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <subscription-id
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         my-sub
      </subscription-id>
      <eventTime>2015-03-09T19:14:56Z</eventTime>
      <datastore-contents xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         <foo>
            <bar>some_string</bar>
         </foo>
      </datastore-contents>
   </notification>
```

Figure 1: Push example

The following is an example of an on-change notification. It contains an update for subscription my-on-change-sub, including a new value for a leaf called beta, which is a child of a top-level container called alpha:

```
<?xml version="1.0" encoding="UTF-8"?>
  <notification
        xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
     <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
           datastore-push:1.0">
        my-on-change-sub
     </subscription-id>
     <eventTime>2015-10-13T12:13:02Z</eventTime>
     <datastore-changes-xml xmlns="urn:ietf:params:xml:ns:netconf:
           datastore-push:1.0">
       <alpha xmlns="http://example.com/yang-push/1.0" >
         <beta>1500</beta>
       </alpha>
    </datastore-changes-xml>
</notification>
```

Figure 2: Push example for on change

The equivalent update when requesting json encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
  <notification
        xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         my-on-change-sub
      </subscription-id>
      <eventTime>2015-10-13T12:13:02Z</eventTime>
      <datastore-changes-json
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
     {
     "ietf-yang-patch:yang-patch": {
         "patch-id": [
             null
         ],
         "edit": [
             {
                 "edit-id": "edit1",
                 "operation": "merge",
                 "target": "/alpha/beta",
                 "value": {
                     "beta": 1500
                 }
             }
         ]
       }
    }
  </datastore-changes-json>
</notification>
```

Figure 3: Push example for on change with JSON

When the beta leaf is deleted, the server may send

```
<?xml version="1.0" encoding="UTF-8"?>
  <notification
        xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
     <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
           datastore-push:1.0">
        my-on-change-sub
     </subscription-id>
     <eventTime>2015-10-13T12:13:02Z</eventTime>
     <datastore-changes-xml xmlns="urn:ietf:params:xml:ns:netconf:
           datastore-push:1.0">
       <alpha xmlns="http://example.com/yang-push/1.0" >
         <beta xc:operation="delete"/>
       </alpha>
    </datastore-changes-xml>
</notification>
```

Figure 4: 2nd push example for on change update

3.7. Subscription management

There are two ways in which subscriptions can be managed: RPC-based, and configuration based.

3.7.1. Subscription management by RPC

RPC-based subscription allows a subscriber to create a subscription via an RPC call. The subscriber and the receiver are the same entity, i.e. a subscriber cannot subscribe or in other ways interfere with a subscription on another receiver's behalf. The lifecycle of the subscription is dependent on the lifecyle of the transport session over which the subscription was requested. For example, when a Netconf session over which a subscription was created is torn down, the subscription is automatically terminated (and needs to be re-initiated when a new session is established). Alternatively, a subscriber can also decide to delete a subscription via another RPC.

When a create-subscription request is successful, the subscription identifier of the freshly created subscription is returned.

A subscription can be rejected for multiple reasons, including the lack of authorization to create a subscription, the lack of read authorization on the requested data node, or the inability of the server to provide a stream with the requested semantics. Rejections trigger the generation of an rpc-reply with an rpc-error element, which indicates why the subscription was rejected and, possibly, negotiation information to facilitate the generation of subscription requests that can be served. The contents of the rpc-error element

follow the specification in [RFC6241]. Datastore-push-specific content is included under <error-info>. When the requester is not authorized to read the requested data node, the returned <error-info> indicates an authorization error and the requested node. For instance, for the following request:

```
<netconf:rpc message-id="101"
   xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <create-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/foo/1.0"
            select="/ex:foo"/>
      <period xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         500
      </period>
      <encoding
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         encode-xml
      </encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 5: Create-Subscription example

the server may return:

```
   <rpc-reply message-id="101"
         xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <rpc-error>
         <error-type>application</error-type>
         <error-tag>access-denied</error-tag>
         <error-severity>error</error-severity>
         <error-info>
            <access-denied xmlns="urn:ietf:params:xml:ns:
                  netconf:datastore-push:1.0">
               <data-node>/ex:foo</data-node>
            </ access-denied >
         </error-info>
      </rpc-error>
   </rpc-reply>
```

Figure 6: Error response example

When the requester is not authorized to execute a subscription request, no <error-info> element should be included in the response.

3.7.2. Subscription management by configuration

Configuration-based subscription allows a subscription to be established as part of a server's configuration. This allows to persist subscriptions. As part of a configured subscription, a receiver needs to be specified. It is thus possible to have a different system acting as subscriber (the client creating the subscription) and as receiver (the client receiving the updates).

3.8. Other considerations

3.8.1. Authorization

A receiver of subscription data may only be sent updates for which they have proper authorization. Data that is being pushed therefore needs to be subjected to a filter that applies all corresponding rules applicable at the time of a specific pushed update, removing any non-authorized data as applicable.

The authorization model for data in YANG datastores is described in the Netconf Access Control Model [RFC6536]. However, some clarifications to that RFC are needed so that the desired access control behavior is applied to pushed updates.

One of these clarifications is that a subscription may only be established if the Receiver has read access to the target data node.

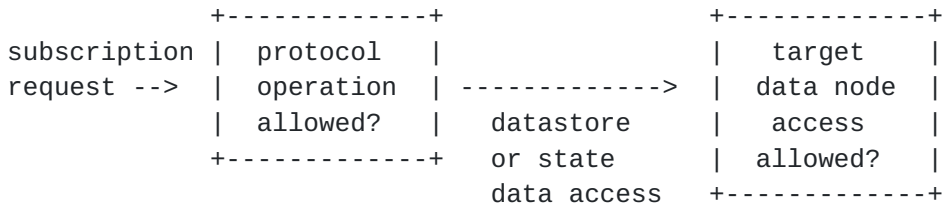

Figure 7: Access control for subscription

Likewise if a receiver no longer has read access permission to a target data node, the subscription must be abnormally terminated (with loss of access permission as the reason provided).

Another clarification to [RFC6536] is that each of the individual nodes in a pushed update must also go through access control filtering. This includes new nodes added since the last push update, as well as existing nodes. For each of these read access must be verified. The methods of doing this efficiently are left to implementation.

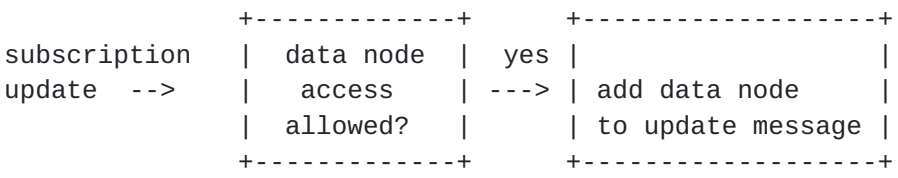

Figure 8: Access control for push updates

If there are read access control changes applied under the target node, no notifications indicating the fact that this has occurred need to be provided.

3.8.2. Additional subscription primitives

Other possible operations include the ability for a Subscriber to request the suspension/resumption of a Subscription with a Publisher. However, subscriber driven suspension is not viewed as essential at this time, as a simpler alternative is to remove a subscription and recreate it when needed.

It should be noted that this does not affect the ability of the Publisher to suspend a subscription. This can occur in cases the server is not able to serve the subscription for a certain period of time, and indicated by a corresponding notification.

3.8.3. Robustness and reliability considerations

Particularly in the case of on-change push updates, it is important that push updates do not get lost. However, datastore-push uses a secure and reliable transport. Notifications are not getting reordered, and in addition contain a time stamp. For those reasons, we believe that additional reliability mechanisms at the application level, such as sequence numbers for push updates, are not required.

3.8.4. Update size and fragmentation considerations

Depending on the subscription, the volume of updates can become quite large. There is no inherent limitation to the amount of data that can be included in a notification. That said, it may not always be practical to send the entire update in a single chunk. Implementations MAY therefore choose, at their discretion, to "chunk" updates and break them out into several update notifications.

3.8.5. Additional data streams

The conceptual data stream introduced in this specification, datastore-push, includes the entire YANG datastore in its scope. It

is conceivable to introduce other data streams with more limited scope, for example:

- operdata-push, a datastream containing all operational (read-only) data of a YANG datastore

- operdata-nocounts-push, a datastream containing all operational (read-only) data with the exception of counters

Those data streams make particular sense for use cases involving service assurance (not relying on operational data), and for use cases requiring on-change update triggers which make no sense to support in conjunction with fast-changing counters. While it is possible to specify subtree filters on datastore-push to the same effect, having those data streams greatly simplifies articulating subscriptions in such scenarios.

3.8.6. Implementation considerations

Implementation specifics are outside the scope of this specification. That said,it should be noted that monitoring of operational state changes inside a system can be associated with significant implementation challenges.

Even periodic retrieval of operational state alone, to be able to push it, can consume considerable system resources. Configuration data may in many cases be persisted in an actual database or a configuration file, where retrieval of the database content or the file itself is reasonably straightforward and computationally inexpensive. However, retrieval of operational data may, depending on the implementation, require invocation of APIs, possibly on an object-by-object basis, possibly involving additional internal interrupts, etc.

For those reasons, if is important for an implementation to understand what subscriptions it can or cannot support. It is far preferrable to decline a subscription request, than to accept it only to result in subsequent failure later.

Whether or not a subscription can be supported will in general be determined by a combination of several factors, including the subscription policy (on-change or periodic, with on-change in general being the more challenging of the two), the period in which to report changes (1 second periods will consume more resources than 1 hour periods), the amount of data in the subtree that is being subscribed to, and the number and combination of other subscriptions that are concurrently being serviced.

When providing access control to every node in a pushed update, it is possible to make and update efficient access control filters for an update. These filters can be set upon subscription and applied against a stream of updates. These filters need only be updated when (a) there is a new node added/removed from the subscribed tree with different permissions than its parent, or (b) read access permissions have been changed on nodes under the target node for the subscriber.

4. A YANG data model for management of datastore push subscriptions

4.1. Overview

The YANG data model for datastore push subscriptions is depicted in the following figure.

```
module: ietf-datastore-push
   +--ro system-streams
   |  +--ro system-stream*   system-stream
   +--rw filters
   |  +--rw filter* [filter-id]
   |     +--rw filter-id         filter-id
   |     +--rw (filter-type)?
   |        +--:(subtree)
   |        |  +--rw subtree-filter?   subtree-filter
   |        +--:(xpath)
   |           +--rw xpath-filter?     yang:xpath1.0
   +--rw subscription-config
   |  +--rw datastore-push-subscription* [subscription-id]
   |     +--rw subscription-id      subscription-id
   |     +--rw target-datastore?    datastore
   |     +--rw stream?              system-stream
   |     +--rw encoding?            encoding
   |     +--rw start-time?          yang:date-and-time
   |     +--rw stop-time?           yang:date-and-time
   |     +--rw (update-trigger)?
   |     |  +--:(periodic)
   |     |  |  +--rw period?              yang:timeticks
   |     |  +--:(on-change)
   |     |     +--rw no-synch-on-start?   empty
   |     |     +--rw dampening-period     yang:timeticks
   |     |     +--rw excluded-change*     change-type
   |     +--rw (filterspec)?
   |     |  +--:(inline)
   |     |  |  +--rw (filter-type)?
   |     |  |     +--:(subtree)
   |     |  |     |  +--rw subtree-filter?      subtree-filter
   |     |  |     +--:(xpath)
   |     |  |        +--rw xpath-filter?       yang:xpath1.0
```

```
      |     |  +--:(by-reference)
      |     |     +--rw filter-ref?          filter-ref
      |     +--rw receiver-address
      |        +--rw (push-base-transport)?
      |           +--:(tcpudp)
      |              +--rw tcpudp
      |                 +--rw address?   inet:host
      |                 +--rw port?      inet:port-number
     +--ro subscriptions
        +--ro datastore-push-subscription* [subscription-id]
           +--ro subscription-id            subscription-id
           +--ro configured-subscription?   empty
           +--ro subscription-status?       identityref
           +--ro target-datastore?          datastore
           +--ro stream?                    system-stream
           +--ro encoding?                  encoding
           +--ro start-time?                yang:date-and-time
           +--ro stop-time?                 yang:date-and-time
           +--ro (update-trigger)?
           |  +--:(periodic)
           |  |  +--ro period?                    yang:timeticks
           |  +--:(on-change)
           |     +--ro no-synch-on-start?         empty
           |     +--ro dampening-period           yang:timeticks
           |     +--ro excluded-change*           change-type
           +--ro (filterspec)?
           |  +--:(inline)
           |  |  +--ro (filter-type)?
           |  |     +--:(subtree)
           |  |     |  +--ro subtree-filter?            subtree-filter
           |  |     +--:(xpath)
           |  |        +--ro xpath-filter?             yang:xpath1.0
           |  +--:(by-reference)
           |     +--ro filter-ref?               filter-ref
           +--ro receiver-address
              +--ro (push-base-transport)?
                 +--:(tcpudp)
                    +--ro tcpudp
                       +--ro address?   inet:host
                       +--ro port?      inet:port-number
```

Figure 9: Model structure

The components of the model are described in the following subsections.

4.2. System streams

Container "system-streams" is used to indicate which data streams are provided by the system and can be subscribed to. For this purpose, it contains a leaf list of data nodes identifying the supported streams.

4.3. Filters

Container "filters" contains a list of configurable data filters, each specified in its own list element. This allows users to configure filters separately from an actual subscription, which can then be referenced from a subscription. This facilitates the reuse of filter definitions, which can be important in case of complex filter conditions.

Two types of filters can be specified as part of a filter list element. Subtree filters follow syntax and semantics of RFC 6241 and allow to specify which subtree(s) to subscribe to. In addition, XPath filters can be specified for more complex filter conditions. If several filters are specified. When both types of filters are included, a logical "and" applies.

It is conceivable to introduce other types of filters; in that case, the data model needs to be augmented accordingly.

4.4. Subscription configuration

Container "subscription-config" allows for the static configuration of subscriptions, i.e. subscriptions that are created via configuration as opposed to RPC. Each subscription is represented through its own list element, including the following components:

- "subscription-id" is an identifier used to refer to the subscription.

- "target-datastore" is used to refer to the datastore the subscription refer to. By default, the datastore will always be "running".

- "stream" refers to the stream being subscribed to. The subscription model assumes the presence of perpetual and continuous streams of updates. Various streams are defined: "push-update" covers the entire set of YANG data in the server. "operational-push" covers all operational data, while "config-push" covers all configuration data. Other streams could be introduced in augmentations to the model by introducing additional identities.

- "encoding" refers to the encoding requested for the data updates. By default, updates are encoded using XML. However, JSON can be requested as an option and other encodings may be supported in the future.

- "start-time" specifies when the subscription is supposed to start. The start time also serves as anchor time for periodic subscriptions (see below).

- "stop-time" specifies a stop time for the subscription. Once the stop time is reached, the subscription is automatically terminated. However, even when terminated, the subscription entry remains part of the configuration unless explicity deleted from the configuration. It is possible to effectively "resume" a stopped subscription by reconfiguring the stop time.

- A choice of subscription policies allows to define when to send new updates - periodic or on change.

  * For periodic subscriptions, the trigger is defined by a "period", a parameter that defines the interval with which updates are to be pushed. The start time of the subscription serves as anchor time, defining one specific point in time at which an update needs to be sent. Update intervals always fall on the points in time that are a multiple of a period after the start time.

  * For on-change subscriptions, the trigger occurs whenever a change in the subscribed information is detected. On-change subscriptions have more complex semantics that is guided by additional parameters. "dampening-period" specifies the interval that must pass before a successive update for the same data node is sent. The first time a change is detected, the update is sent immediately. If a subsequent change is detected, another update is only sent once the dampening period has passed, containing the value of the data node that is then valid. "excluded-change" allows to restrict the types of changes for which updates are sent (changes to object values, object creation or deletion events). "no-synch-on-start" is a flag that allows to specify whether or not a complete update with all the subscribed data should be sent at the beginning of a subscription; if the flag is omitted, a complete update is sent to facilitate synchronization. It is conceivable to augment the data model with additional parameters in the future to specify even more refined policies, such as parameters that specify the magnitude of a change that must occur before an update is triggered.

- Filters for a subscription can be specified using a choice, allowing to either reference a filter that has been separately configured or entering its definition inline.

- Finally, a receiver for the subscription can be specified. The receiver does not have to be the same system that configures the subscription.

It should be noted that a subscription created through configuration cannot be deleted using an RPC. Likewise, subscriptions created through RPC cannot be deleted through configuration.

4.5. Subscription monitoring

Subscriptions can be subjected to management themselves. For example, it is possible that a server may no longer be able to serve a subscription that it had previously accepted. Perhaps it has run out of resources, or internal errors may have occurred. When this is the case, a server needs to be able to temporarily suspend the subscription, or even to terminate it. More generally, the server should provide a means by which the status of subscriptions can be monitored.

Container "subscriptions", contains operational data for all subscriptions that are currently active. This includes subscriptions that were created using RPC, as well as subscriptions created as part of the configuration when current time is between start and stop time.

Each subscription is represented as a list element "datastore-push-subscription". The associated information includes an identifier for the subscription, a subscription status, as well as the various subscription parameters that are in effect. The subscription status indicates whether the subscription is currently active and healthy, or if it is degraded in some form.

Subscriptions are automatically removed from the list once they expire (reaching stop-time)or are terminated, whether through RPC or deletion from the configuration.

4.6. Notifications

A server needs to indicate any changes in status of a subscription to the receiver through a notification. Specifically, subscribers need to be informed of the following:

- A subscription has been temporarily suspended (including the reason)

- A subscription (that had been suspended earlier) is once again operational

- A subscription has been terminated (including the reason)

- A subscription has been modified (including the current set of subscription parameters in effect)

Finally, a server might provide additional information about subscriptions, such as statistics about the number of data updates that were sent. However, such information is currently outside the scope of this specification.

4.7. RPCs

Yang-push subscriptions are created, modified, and deleted using three RPCs.

4.7.1. Create-subscription RPC

The subscriber sends a create-subscription RPC with the parameters in section 3.1. For instance

```
<netconf:rpc message-id="101"
      xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <create-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/foo/1.0"
            select="/ex:foo"/>
      <period xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         500
      </period>
      <encoding xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         encode-xml
      </encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 10: Create-subscription RPC

The server must respond explicitly positively (i.e., subscription accepted) or negatively (i.e., subscription rejected) to the request. Positive responses include the subscription-id of the accepted subscription. In that case a server may respond:

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

Figure 11: Create-subscription positive RPC response

A subscription can be rejected for multiple reasons, including the lack of authorization to create a subscription, the lack of read authorization on the requested data node, or the inability of the server to provide a stream with the requested semantics. Rejections trigger the generation of an rpc-reply with an rpc-error element, which indicates why the subscription was rejected and, possibly, negotiation information to facilitate the generation of subscription requests that can be served. The contents of the rpc-error element follow the specification in [RFC6241]. Datastore-push-specific content is included under <error-info>.

When the requester is not authorized to read the requested data node, the returned <error-info> indicates an authorization error and the requested node. For instance, if the above request was unauthorized to read node "ex:foo" the server may return:

```
<rpc-reply message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <rpc-error>
      <error-type>application</error-type>
      <error-tag>access-denied</error-tag>
      <error-severity>error</error-severity>
      <error-info>
         <access-denied xmlns="urn:ietf:params:xml:ns:
               netconf:datastore-push:1.0">
            <data-node>/ex:foo</ data-node >
         </ access-denied >
      </error-info>
   </rpc-error>
</rpc-reply>
```

Figure 12: Create-subscription access denied response

When the requester is not authorized to execute a subscription request, no <error-info> element should be included in the response.

If a request is rejected because the server is not capable to serve it, the server SHOULD include in the returned error what request parameters were not supported and what subscription parameters would have been accepted for the request. This information is included in the <error-info>, which is split into two sections. First, <unsupported-parameters>, which includes the parameters in the request the server cannot serve. Second <supported-subscription>, which constitute suggestions that, when followed; increase the likelihood of success for subsequent requests. However, they are no guarantee that subsequent requests for this client or others will in fact be accepted.

For example, for the following request:

```
<netconf:rpc message-id="101"
      xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <create-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/foo/1.0"
            select="/ex:foo"/>
      <dampening-period
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         10
      </dampening-period>
      <encoding
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         encode-xml
      </encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 13: Create-subscription request example 2

A server that cannot serve on-change updates may return the following:

```
<rpc-reply message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <rpc-error>
      <error-type>application</error-type>
      <error-tag>operation-not-supported</error-tag>
      <error-severity>error</error-severity>
      <error-info>
         <unsupported-parameters xmlns="urn:ietf:params:xml:ns:
               netconf:datastore-push:1.0">
            <dampening-period
                  xmlns="urn:ietf:params:xml:ns:netconf:
                  datastore-push:1.0">
               10
            </dampening-period>
         </unsupported-parameters >
         <supported-subscription xmlns="urn:ietf:params:xml:ns:
               netconf:datastore-push:1.0">
            <period>3000</period>
         </supported-subscription>
      </error-info>
   </rpc-error>
</rpc-reply>
```

Figure 14: Create-subscription error response example 2

4.7.2. Modify-subscription RPC

The subscriber may send a modify-subscription for a previously accepted subscription that has not been deleted. The subscriber may change any subscription parameters by including the new values in the modifys-subscription rpc. Parameters not included in the rpc should remain unmodified. For illustration purposes we include an exchange example where a subscriber modifies the period of the subscription.

```
<netconf:rpc message-id="101"
   xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <modify-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <stream>push-update</stream>
      <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         my-sub
      </subscription-id>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/foo/1.0"
            select="/ex:foo"/>
      <period
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         500
      </period>
      <encoding
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         encode-xml
      </encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 15: Modify subscription request

The server must respond explicitly positively (i.e., subscription accepted) or negatively (i.e., subscription rejected) to the request. Positive responses include the subscription-id of the accepted subscription. In that case a server may respond:

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


<netconf:rpc message-id="102"
      xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <modify-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         my-sub
      </subscription-id>
      <period xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         100
      </period>
      <encoding
            xmlns="urn:ietf:params:xml:ns:netconf:datastore-push:1.0">
         encode-xml
      </encoding>
   </modify-subscription>
</netconf:rpc>

<rpc-reply message-id="102"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
         datastore-push:1.0">
      my-sub
   </subscription-id>
</rpc-reply>
```

Figure 16: Modify subscription response

If the subscription modification is rejected, the server must send a response like it does for a create-subscription and maintain the subscription as it was before the modification request. A subscription may be modified multiple times.

4.7.3. Delete-subscription RPC

To stop receiving updates from a subscription and effectively eliminate the subscription, it can send a delete-subscription RPC, which takes as only input the subscription-id. For example

```
<netconf:rpc message-id="103"
      xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
   <delete-subscription
         xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <subscription-id xmlns="urn:ietf:params:xml:ns:netconf:
            datastore-push:1.0">
         my-sub
      </subscription-id>
   </delete-subscription>
</netconf:rpc>

<rpc-reply message-id="103"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <ok/>
</rpc-reply>
```

Figure 17: Delete subscription

5. YANG module

```
<CODE BEGINS>
file "ietf-datastore-push@2015-10-09.yang"

module ietf-datastore-push {
  namespace "urn:ietf:params:xml:ns:yang:ietf-datastore-push";
  prefix yp;

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

  organization "IETF";
  contact
    "WG Web:   <http://tools.ietf.org/wg/netconf/>
     WG List:  <mailto:netconf@ietf.org>

     WG Chair: Mahesh Jethanandani
               <mailto:mjethanandani@gmail.com>
```

```
        WG Chair: Mehmet Ersue
                  <mailto:mehmet.ersue@nokia.com>

        Editor:   Alexander Clemm
                  <mailto:alex@cisco.com>

        Editor:   Alberto Gonzalez Prieto
                  <mailto:albertgo@cisco.com>

        Editor:   Eric Voit
                  <mailto:evoit@cisco.com>";
    description
      "This module contains conceptual YANG specifications
      for datastore push.";

    revision 2015-10-13 {
      description
        "Initial revision.";
      reference
        "YANG Datastore Push, draft-clemm-netconf-yang-push-02";
    }

    feature on-change {
      description
        "This feature indicates that on-change updates are supported.";
    }

    feature json {
      description
        "This feature indicates that JSON encoding of push updates is
        supported.";
    }

    identity subscription-stream-status {
      description
        "Base identity for the status of subscriptions and
        datastreams.";
    }

    identity active {
      base subscription-stream-status;
      description
        "Status is active and healthy.";
    }

    identity inactive {
      base subscription-stream-status;
      description
```

```
        "Status is inactive, for example outside the
        interval between start time and stop time.";
    }

    identity in-error {
      base subscription-stream-status;
      description
        "The status is in error or degraded, meaning that
        stream and/or subscription are currently unable to provide
        the negotiated updates.";
    }

    identity subscription-errors {
      description
        "Base identity for subscription errors.";
    }

    identity internal-error {
      base subscription-errors;
      description
        "Subscription failures caused by server internal error.";
    }

    identity no-resources {
      base subscription-errors;
      description
        "Lack of resources, e.g. CPU, memory, bandwidth";
    }

    identity subscription-deleted {
      base subscription-errors;
      description
        "The subscription was terminated because the subscription
        was deleted.";
    }

    identity other {
      base subscription-errors;
      description
        "Fallback reason - any other reason";
    }

    identity encodings {
      description
        "Base identity to represent data encodings";
    }

    identity encode-xml {
```

```
      base encodings;
      description
        "Encode data using XML";
    }

    identity encode-json {
      base encodings;
      description
        "Encode data using JSON";
    }

    identity system-streams {
      description
        "Base identity to represent a conceptual system-provided
        datastream of datastore updates with predefined semantics.";
    }

    identity datastore-push {
      base system-streams;
      description
        "A conceptual datastream consisting of all datastore
        updates, including operational and configuration data.";
    }

    identity operational-push {
      base system-streams;
      description
        "A conceptual datastream consisting of updates of all
        operational data.";
    }

    identity config-push {
      base system-streams;
      description
        "A conceptual datastream consisting of updates of all
        configuration data.";
    }

    identity datastore {
      description
        "An identity that represents a datastore.";
    }

    identity running {
      base datastore;
      description
        "Designates the running datastore";
    }
```

```
identity startup {
  base datastore;
  description
    "Designates the startup datastore";
}

typedef datastore-contents-xml {
  type string;
  description
    "This type is be used to represent datastore contents,
    i.e. a set of data nodes with their values, in XML.
    The syntax corresponds to the syntax of the data payload
    returned in a corresponding Netconf get operation with the
    same filter parameters applied.";
  reference "RFC 6241 section 7.7";
}

typedef datastore-changes-xml {
  type string;
  description
    "This type is used to represent a set of changes in a
    datastore encoded in XML, indicating for datanodes whether
    they have been created, deleted, or updated.  The syntax
    corresponds to the syntax used to when editing a
    datastore using the edit-config operation in Netconf.";
  reference "RFC 6241 section 7.2";
}

typedef datastore-contents-json {
  type string;
  description
    "This type is be used to represent datastore contents,
    i.e. a set of data nodes with their values, in JSON.
    The syntax corresponds to the syntax of the data
    payload returned in a corresponding RESTCONF get
    operation with the same filter parameters applied.";
  reference "RESTCONF Protocol";
}

typedef datastore-changes-json {
  type string;
  description
    "This type is used to represent a set of changes in a
    datastore encoded in JSON, indicating for datanodes whether
    they have been created, deleted, or updated.  The syntax
    corresponds to the syntax used to patch a datastore
    using the yang-patch operation with Restconf.";
  reference "draft-ietf-netconf-yang-patch";
```

```
    }

    typedef filter-id {
      type string;
      description
        "This type defines an identifier for a filter.";
    }

    typedef subtree-filter {
      type string;
      description
        "This type is used to specify the subtree that the
        subscription refers to. Its syntax follows the subtree
        filter syntax specified for Netconf in RFC 6241,
        section 6.";
      reference "RFC 6241 section 6";
    }

    typedef datastore {
      type identityref {
        base datastore;
      }
      description
        "Used to refer to a datastore, for example, to running";
    }

    typedef subscription-id {
      type string {
        length "1 .. max";
      }
      description
        "A client-provided identifier for the subscription.";
    }

    typedef subscription-term-reason {
      type identityref {
        base subscription-errors;
      }
      description
        "Reason for a server to terminate a subscription.";
    }

    typedef subscription-susp-reason {
      type identityref {
        base subscription-errors;
      }
      description
        "Reason for a server to suspend a subscription.";
```

```
    }

    typedef encoding {
      type identityref {
        base encodings;
      }
      description
        "Specifies a data encoding, e.g. for a data subscription.";
    }

    typedef change-type {
      type enumeration {
        enum "create" {
          description
            "A new data node was created";
        }
        enum "delete" {
          description
            "A data node was deleted";
        }
        enum "modify" {
          description
            "The value of a data node has changed";
        }
      }
      description
        "Specifies different types of changes that may occur
        to a datastore.";
    }

    typedef system-stream {
      type identityref {
        base system-streams;
      }
      description
        "Specifies a system-provided datastream.";
    }

    typedef filter-ref {
      type leafref {
        path "/yp:filters/yp:filter/yp:filter-id";
      }
      description
        "This type is used to reference a yang push filter.";
    }

    grouping datatree-filter {
      description
```

```
        "This grouping defines filters for a datastore tree.";
      choice filter-type {
        description
          "A filter needs to be a single filter of a given type.
          Mixing and matching of multiple filters does not occur
          at the level of this grouping.";
        case subtree {
          description
            "Subtree filter";
          leaf subtree-filter {
            type subtree-filter;
            description
              "Datastore subtree of interest.";
          }
        }
        case xpath {
          description
            "XPath filter";
          leaf xpath-filter {
            type yang:xpath1.0;
            description
              "Xpath defining the data items of interest.";
          }
        }
      }
    }

    grouping subscription-info {
      description
        "This grouping describes basic information concerning a
         subscription.";
      leaf target-datastore {
        type datastore;
        default "running";
        description
          "The datastore that is the target of the subscription.
          If not specified, running applies.";
      }
      leaf stream {
        type system-stream;
        default "datastore-push";
        description
          "The name of the stream subscribed to.";
      }
      leaf encoding {
        type encoding;
        default "encode-xml";
        description
```

```
          "The type of encoding for the subscribed data.
          Default is XML";
        }
        leaf start-time {
          type yang:date-and-time;
          description
            "Designates the time at which a subscription is supposed
            to start, or immediately, in case the start-time is in
            the past. For periodic subscription, the start time also
            serves as anchor time from which the time of the next
            update is computed. The next update will take place at the
            next period interval from the anchor time.
            For example, for an anchor time at the top of a minute
            and a period interval of a minute, the next update will
            be sent at the top of the next minute.";
        }
        leaf stop-time {
          type yang:date-and-time;
          description
            "Designates the time at which a subscription will end.
            When a subscription reaches its stop time, it will be
            automatically deleted.";
        }
        choice update-trigger {
          description
            "Defines necessary conditions for sending an event  to
             the subscriber.";
          case periodic {
            description
              "The agent is requested to notify periodically the
              current values of the datastore or the subset
              defined by the filter.";
            leaf period {
              type yang:timeticks;
              description
                "Elapsed time between notifications.";
            }
          }
          case on-change {
            description
              "The agent is requested to notify changes in
               values in the datastore or a subset of it defined
               by a filter.";
            leaf no-synch-on-start {
              type empty;
              description
                "This leaf acts as a flag that determines behavior at the
                start of the subscription.  When present,
```

```
            synchronization of state at the beginning of the
            subscription is outside the scope of the subscription.
            Only updates about changes that are observed from the
            start time, i.e. only push-change-update notifications
            are sent.
            When absent (default behavior), in order to facilitate
            a receiver's synchronization, a full update is sent
            when the subscription starts using a push-update
            notification, just like in the case of a periodic
            subscription.  After that, push-change-update
            notifications are sent.";
        }
        leaf dampening-period {
          type yang:timeticks;
          mandatory true;
          description
            "Minimum amount of time that needs to have
            passed since the last time an update was
            provided.";
        }
        leaf-list excluded-change {
          type change-type;
          description
            "Use to restrict which changes trigger an update.
            For example, if modify is excluded, only creation and
            deletion of objects is reported.";
        }
      }
    }
    choice filterspec {
      description
        "Filter can be specified in-line, as part of the
        subscription, or configured separately and referenced
        here.  If no filter is specified, the entire datatree
        is of interest.";
      case inline {
        description
          "Filter is defined as part of the subscription.";
        uses datatree-filter;
      }
      case by-reference {
        description
          "Incorporate a filter that has been configured
          separately.";
        leaf filter-ref {
          type filter-ref;
          description
            "References the filter to incorporate for the
```

```
            subscription.";
        }
      }
    }
  }

  grouping receiver-info {
    description
      "Defines a reusable snippet that defines the address of the
      intended receiver of push updates for a subscription.";
    container receiver-address {
      description
        "This container contains the address information of the
         receiver.";
      choice push-base-transport {
        description
          "This choice can be augmented with different options,
           depending on the transport underlying the push
           transport.";
        case tcpudp {
          description
            "For Netconf and Restconf, TCP is the base transport.";
          container tcpudp {
            description
              "Contains TCP / UDP addressing information";
            leaf address {
              type inet:host;
              description
                "The leaf uniquely specifies the address of the
                 remote host. One of the following must be
                 specified: an ipv4 address, an ipv6 address,
                 or a host name.";
            }
            leaf port {
              type inet:port-number;
              description
                "This leaf specifies the port number used to
                 deliver messages to the remote server.";
            }
          }
        }
      }
    }
  }

  rpc create-subscription {
    description
      "This RPC allows a subscriber to create a subscription
```

```
        on its own behalf.  If successful, the subscription
        remains in effect for the duration of the subscriber's
        association with the publisher, or until the subscription
        is terminated by virtue of a delete-subscription request.";
      input {
        uses subscription-info;
      }
      output {
        leaf subscription-id {
          type subscription-id;
          description
            "Identifier used for this subscription.";
        }
      }
    }
    rpc modify-subscription {
      description
        "This RPC allows a subscriber to modify a subscription
        that was previously created using create-subscription.
        If successful, the subscription
        remains in effect for the duration of the subscriber's
        association with the publisher, or until the subscription
        is terminated by virtue of a delete-subscription request.";
      input {
        leaf subscription-id {
          type subscription-id;
          description
            "Identifier to use for this subscription.";
        }
      }
    }
    rpc delete-subscription {
      description
        "This RPC allows a subscriber to delete a subscription that
        was previously created using create-subscription.";
      input {
        leaf subscription-id {
          type subscription-id;
          description
            "Identifier of the subscription that is to be deleted.
            Only subscriptions that were created using
            create-subscription can be deleted via this RPC.";
        }
      }
    }
    notification push-update {
      description
        "This notification contains a periodic push update.
```

```
        This notification shall only be sent to receivers
        of a subscription; it does not constitute a general-purpose
        notification.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the subscription because of which the
          notification is sent.";
      }
      leaf time-of-update {
        type yang:date-and-time;
        description
          "This leaf contains the time of the update.";
      }
      choice encoding {
        description
          "Distinguish between the proper encoding that was specified
          for the subscription";
        case encode-xml {
          description
            "XML encoding";
          leaf datastore-contents-xml {
            type datastore-contents-xml;
            description
              "This contains data encoded in XML,
              per the subscription.";
          }
        }
        case encode-json {
          if-feature json;
          description
            "JSON encoding";
          leaf datastore-contents-json {
            type datastore-contents-json;
            description
              "This leaf contains data encoded in JSON,
              per the subscription.";
          }
        }
      }
    }
    notification push-change-update {
      description
        "This notification contains an on-change push update.
        This notification shall only be sent to the receivers
        of a subscription; it does not constitute a general-purpose
        notification.";
```

```
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the subscription because of which the
          notification is sent.";
      }
      leaf time-of-update {
        type yang:date-and-time;
        description
          "This leaf contains the time of the update, i.e. the
          time at which the change was observed.";
      }
      choice encoding {
        description
          "Distinguish between the proper encoding that was specified
          for the subscription";
        case encode-xml {
          description
            "XML encoding";
          leaf datastore-changes-xml {
            type datastore-changes-xml;
            description
              "This contains datastore contents that has changed
              since the previous update, per the terms of the
              subscription.  Changes are encoded analogous to
              the syntax of a corresponding Netconf edit-config
              operation.";
          }
        }
        case encode-json {
          if-feature json;
          description
            "JSON encoding";
          leaf datastore-changes-yang {
            type datastore-changes-json;
            description
              "This contains datastore contents that has changed
              since the previous update, per the terms of the
              subscription.  Changes are encoded analogous
              to the syntax of a corresponding RESTCONF yang-patch
              operation.";
          }
        }
      }
    }
    notification subscription-started {
      description
```

```
        "This notification indicates that a subscription has
        started and data updates are beginning to be sent.
        This notification shall only be sent to receivers
        of a subscription; it does not constitute a general-purpose
        notification.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the affected subscription.";
      }
      uses subscription-info;
    }
    notification subscription-suspended {
      description
        "This notification indicates that a suspension of the
        subscription by the server has occurred.  No further
        datastore updates will be sent until subscription
        resumes.
        This notification shall only be sent to receivers
        of a subscription; it does not constitute a general-purpose
        notification.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the affected subscription.";
      }
      leaf reason {
        type subscription-susp-reason;
        description
          "Provides a reason for why the subscription was
          suspended.";
      }
    }
    notification subscription-resumed {
      description
        "This notification indicates that a subscription that had
        previously been suspended has resumed. Datastore updates
        will once again be sent.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the affected subscription.";
      }
    }
    notification subscription-modified {
```

```
      description
        "This notification indicates that a subscription has
        been modified.  Datastore updates sent from this point
        on will conform to the modified terms of the
        subscription.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the affected subscription.";
      }
      uses subscription-info;
    }
    notification subscription-terminated {
      description
        "This notification indicates that a subscription has been
        terminated.";
      leaf subscription-id {
        type subscription-id;
        mandatory true;
        description
          "This references the affected subscription.";
      }
      leaf reason {
        type subscription-term-reason;
        description
          "Provides a reason for why the subscription was
          terminated.";
      }
    }
    container system-streams {
      config false;
      description
        "This container contains a leaf list of built-in
         streams that are provided by the system.";
      leaf-list system-stream {
        type system-stream;
        description
          "Identifies a built-in stream that is supported by the
          system.  Streams are associated with their own identities,
          each of which carries a special semantics.";
      }
    }
    container filters {
      description
        "This container contains a list of configurable filters
        that can be applied to subscriptions.  This facilitates
        the reuse of complex filters once defined.";
```

```
      list filter {
        key "filter-id";
        description
          "A list of configurable filters that can be applied to
          subscriptions.";
        leaf filter-id {
          type filter-id;
          description
            "An identifier to differentiate between filters.";
        }
        uses datatree-filter;
      }
    }
    container subscription-config {
      description
        "Contains the list of subscriptions that are configured,
         as opposed to established via RPC or other means.";
      list datastore-push-subscription {
        key "subscription-id";
        description
          "Content of a yang-push subscription.";
        leaf subscription-id {
          type subscription-id;
          description
            "Identifier to use for this subscription.";
        }
        uses subscription-info;
        uses receiver-info;
      }
    }
    container subscriptions {
      config false;
      description
        "Contains the list of currently active subscriptions,
        i.e. subscriptions that are currently in effect,
        used for subscription management and monitoring purposes.
        This includes subscriptions that have been setup via RPC
        primitives, e.g. create-subscription, delete-subscription,
        and modify-subscription, as well as subscriptions that
        have been established via configuration.";
      list datastore-push-subscription {
        key "subscription-id";
        config false;
        description
          "Content of a yang-push subscription.
          Subscriptions can be created using a control channel
          or RPC, or be established through configuration.";
        leaf subscription-id {
```

```
          type subscription-id;
          description
            "Identifier of this subscription.";
        }
        leaf configured-subscription {
          type empty;
          description
            "The presence of this leaf indicates that the
            subscription originated from configuration, not through
            a control channel or RPC.";
        }
        leaf subscription-status {
          type identityref {
            base subscription-stream-status;
          }
          description
            "The status of the subscription.";
        }
        uses subscription-info;
        uses receiver-info;
      }
    }
  }
  <CODE ENDS>
```

6. Security Considerations

Subscriptions could be used to attempt to overload servers of YANG datastores. For this reason, it is important that the server has the ability to decline a subscription request if it would deplete its resources. In addition, a server needs to be able to suspend an existing subscription when needed. When this occur, the subscription status is updated accordingly and the clients are notified. Likewise, requests for subscriptions need to be properly authorized.

A subscription could be used to retrieve data in subtrees that a client has not authorized access to. Therefore it is important that data pushed based on subscriptions is authorized in the same way that regular data retrieval operations are. Data being pushed to a client needs therefore to be filtered accordingly, just like if the data were being retrieved on-demand. The Netconf Authorization Control Model applies.

A subscription could be configured on another receiver's behalf, with the goal of flooding that receiver with updates. One or more publishers could be used to overwhelm a receiver which doesn't even support subscriptions. Clients which do not want pushed data need only terminate or refuse any transport sessions from the publisher.

7. Acknowledgments

We wish to acknowledge the helpful contributions, comments, and suggestions that were received from Ambika Prasad Tripathy and Einar Nilsen-Nygaard.

8. References

8.1. Normative References

[RFC1157] Case, J., Fedor, M., Schoffstall, M., and J. Davin, "Simple Network Management Protocol (SNMP)", RFC 1157, DOI 10.17487/RFC1157, May 1990, <http://www.rfc-editor.org/info/rfc1157>.

[RFC5277] Chisholm, S. and H. Trevino, "NETCONF Event Notifications", RFC 5277, July 2008.

[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>.

[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>.

[RFC6470] Bierman, A., "Network Configuration Protocol (NETCONF) Base Notifications", RFC 5277, February 2012.

[RFC6536] Bierman, A. and M. Bjorklund, "Network Configuration Protocol (NETCONF) Access Control Model", RFC 6536, DOI 10.17487/RFC6536, March 2012, <http://www.rfc-editor.org/info/rfc6536>.

8.2. Informative References

[I-D.clemm-netmod-mount] Clemm, A., Medved, J., and E. Voit, "Mounting YANG-defined information from remote datastores", draft-clemm-netmod-mount-03 (work in progress), April 2015.

[I-D.i2rs-pub-sub-requirements] Voit, E., Clemm, A., and A. Gonzalez Prieto, "Requirements for Subscription to YANG Datastores", draft-ietf-i2rs-pub-sub-requirements-03 (work in progress), October 2015.

[I-D.ietf-netconf-restconf]
Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", I-D draft-ietf-netconf-restconf-07, July 2015.

[I-D.ietf-netconf-yang-patch]
Bierman, A., Bjorklund, M., and K. Watsen, "YANG Patch Media Type", draft-ietf-netconf-yang-patch-05 (work in progress), July 2015.

[I-D.ietf-netmod-yang-json]
Lhotka, L., "JSON Encoding of Data Modeled with YANG", draft-ietf-netmod-yang-json-06 (work in progress), October 2015.

[I-D.voit-netmod-peer-mount-requirements]
Voit, E., Clemm, A., and S. Mertens, "Requirements for Peer Mounting of YANG subtrees from Remote Datastores", draft-voit-netmod-peer-mount-requirements-03 (work in progress), September 2015.

Authors' Addresses

Alexander Clemm
Cisco Systems

EMail: alex@cisco.com

Alberto Gonzalez Prieto
Cisco Systems

EMail: albertgo@cisco.com

Eric Voit
Cisco Systems

EMail: evoit@cisco.com