Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 26, 2016

A. Clemm
A. Gonzalez Prieto
E. Voit
A. Tripathy
E. Nilsen-Nygaard
Cisco Systems
February 23, 2016

Subscribing to YANG datastore push updates
draft-ietf-netconf-yang-push-01.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 a receiver 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 dynamically subscribe to updates of a YANG datastore and that allows the server to push those updates. Additionally, support for static subscriptions configured directly on the server are also useful when dynamic signaling is undesirable or unsupported. 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 dynamically 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.

- An alternative mechanism that allows for the static configuration of subscriptions to automatic data store updates as part of a device configuration. In addition to the aspects that are configurable for a dynamic subscription (filter criteria, update policy), static configuration of subscriptions allows for the definition of additional aspects such as intended receivers and alternative transport options.

- 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.

Dynamic subscription: A subscription negotiated between subscriber and publisher via create, modify, and delete RPCs respectively control plane signaling messages that are part of an existing management association between and publisher. Subscriber and receiver are the same system.

NACM: NETCONF Access Control Model

NETCONF: Network Configuration Protocol

Publisher: A server that sends push updates to a receiver according to the terms of a subscription. In general, the publisher is also the "owner" of the subscription.

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

Receiver: The target of push updates of a subscription. In case of a dynamic subscription, receiver and subscriber are the same system. However, in the case of a static subscription, the receiver may be a different system than the one that configured the subscription.

RPC: Remote Procedure Call

SNMP: Simple Network Management Protocol

Static subscription: A subscription installed as part of a configuration datastore via a configuration interface.

Subscriber: A client that negotiates a subscription with a server ("publisher"). A client that creates a static subscription is also considered a subscriber, even if it is not necessarily the receiver of a subscription.

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 for a subscription service, which supports the dynamic as well as static creation of subscriptions to information updates of operational or configuration YANG data which are subsequently pushed from the server to the client.

Dynamic subscriptions are initiated by clients who want to receive push updates. Servers respond to requests for the creation of subscriptions 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. Similarly, static subscriptions are configured as part of a device's configuration. Once a subscription has been established, datastore push updates are pushed from the server to the receiver 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 for both dynamic and static subscriptions 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. Three types of filters are supported: subtree filters, with the same semantics as defined in [RFC 6241], XPath filters, and RFC 5277 filter, with the same semantics as defined in [RFC 5277]. 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.

In addition, for the configuration of static subscriptions, the following parameters are supported:

- One or more receiver IP addresses (and corresponding ports) intended as the destination for push updates for each subscription. In addition the transport protocol for each destination may be defined.

- Optional parameters to identify an egress interface or IP address / VRF where a subscription updates should be pushed from the publisher.

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" and "update-policy" groupings. Receiver information is defined in the "receiver-info" grouping. Information about the source address is defined in the "push-source-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. Values sent at the end of the dampening period are the values current when that dampening period expires. 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.

- RFC5277 filter: A filter that allows for matching of update notification records per RFC 5277.

Only a single filter can be applied to a subscription at a time.

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 to consider is therefore that of a notification. There are however some specifics that need to be considered. Contrary to other notifications that are associated with alarms and unexpected event occurrences, push updates are solicited, i.e. tied to a particular subscription which triggered the notification, and arguably only of interest to the subscriber, respectively the intended receiver of the subscription. A

subscription therefore needs to be able to distinguish between streams that underlie push updates and streams of other notifications. By the same token, notifications associated with updates and subscriptions to updates need to be distinguished from other notifications, in that they enter a datastream of push updates, not a stream of other event notifications.

A push update 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 define 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". Those notifications define how mechanisms that carry YANG notifications (e.g. Netconf notifications and Restconf) can be used to carry data records with updates of datastore contents as specified by a subscription. It is possible also map notifications to other transports and encodings and use the same subscription model; however, the definition of such mappings is outside the scope of this document.

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 1011, including a subtree with root foo that contains a leaf, bar:

```
<notification
      xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
   <eventTime>2015-03-09T19:14:56Z</eventTime>
   <push-update xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     <subscription-id>1011</subscription-id>
     <time-of-update>2015-03-09T19:14:56Z</time-of-update>
     <datastore-contents-xml>
        <foo>
           <bar>some_string</bar>
        </foo>
     </datastore-contents-xml>
   </push-update>
</notification>
```

Figure 1: Push example

The following is an example of an on-change notification. It contains an update for subscription 89, including a new value for a leaf called beta, which is a child of a top-level container called alpha:

```
<notification
      xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
   <eventTime>2015-03-09T19:14:56Z</eventTime>
   <push-change-update xmlns=
       "urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     <subscription-id>89</subscription-id>
     <time-of-update>2015-03-09T19:14:56Z</time-of-update>
     <datastore-changes-xml>
       <alpha xmlns="http://example.com/sample-data/1.0" >
         <beta>1500</beta>
       </alpha>
     </datastore-changes-xml>
   </push-change-update>
</notification>
```

Figure 2: Push example for on change

The equivalent update when requesting json encoding:

```
<notification
      xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
   <eventTime>2015-03-09T19:14:56Z</eventTime>
   <push-change-update xmlns=
       "urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     <subscription-id>89</subscription-id>
     <time-of-update>2015-03-09T19:14:56Z</time-of-update>
     <datastore-changes-json>
       {
        "ietf-yang-patch:yang-patch": {
        "patch-id": [
          null
        ],
        "edit": [
          {
              "edit-id": "edit1",
              "operation": "merge",
              "target": "/alpha/beta",
              "value": {
                  "beta": 1500
              }
          }
        ]
       }
     }
     </datastore-changes-json>
   </push-change-update>
</notification>
```

Figure 3: Push example for on change with JSON

When the beta leaf is deleted, the server may send

```
<notification
      xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
   <eventTime>2015-03-09T19:14:56Z</eventTime>
   <push-change-update xmlns=
       "urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     <subscription-id>89</subscription-id>
     <time-of-update>2015-03-09T19:14:56Z</time-of-update>
     <datastore-changes-xml>
       <alpha xmlns="http://example.com/sample-data/1.0" >
         <beta urn:ietf:params:xml:ns:netconf:base:1.0:
             operation="delete"/>
       </alpha>
     </datastore-changes-xml>
   </push-change-update>
</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. Any given subscription is either RPC-based or configuration-based. There is no mixing-and-matching of RPC and configuration operations. Specifically, a configured subscription cannot be modified or deleted using RPC. Likewise, a subscription created via RPC cannot be through configuration operations.

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. In such cases, no subscription is created. Instead, the subscription-result

with the failure reason is returned as part of the RPC response. In addition, a set of alternative subscription parameters MAY be returned that would likely have resulted in acceptance of the subscription request, which the subscriber may try for a future subscription attempt.

It should be noted that a rejected subscription does not result in the generation of an rpc-reply with an rpc-error element, as neither the specification of YANG-push specific errors nor the specification of additional data parameters to be returned in an error case are supported as part of a YANG data model.

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:yang:ietf-yang-push:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/sample-data/1.0"
            select="/ex:foo"/>
      <period>500</period>
      <encoding>encode-xml</encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 5: Create-Subscription example

the server might return:

```
<rpc-reply message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <subscription-result
       xmlns="http://urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     error-insufficient-resources
   </subscription-result>
   <period>2000</period>
</rpc-reply>
```

Figure 6: Error response example

A subscriber that creates a subscription using RPC can modify or delete the subscription using other RPCs. When the session between subscriber and publisher is terminated, the subscription is implicitly deleted.

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. Persisted subscriptions allow for a number of additional options than RPC-based 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). In addition, a configured subscription allows to specify which transport protocol should be used, as well as the sender source (for example, a particular interface or an address of a specific VRF) from which updates are to be pushed.

Configuration-based subscriptions cannot be modified or deleted using RPCs. Instead, configured subscriptions are deleted as part of regular configuration operations. Servers SHOULD reject attempts to modify configurations of active subscriptions. This way, race conditions in which a receiver may not be aware of changed subscription policies are avoided.

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.

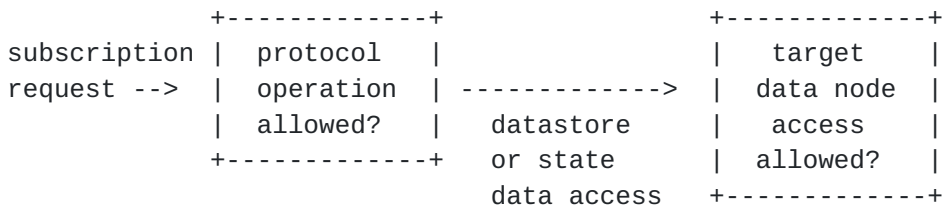

Figure 7: Access control for subscription

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.

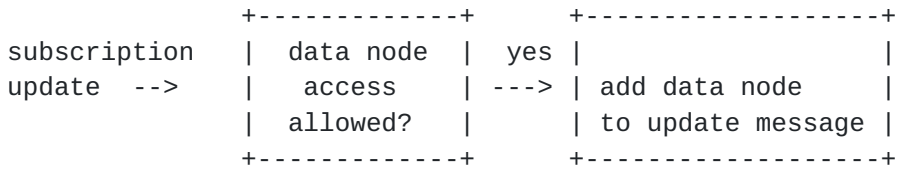

Figure 8: Access control for push updates

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.

Yang-push uses a secure and reliable transport. Notifications are not getting reordered, and in addition contain a time stamp. For those reasons, for the transport of push-updates, we believe that additional reliability mechanisms at the application level, such as sequence numbers for push updates, are not required.

At the same time, it is conceivable that under certain circumstances, a push server is not able to generate the update notifications that it had committed to when accepting a subcription. In those circumstances, the server needs to inform the receiver of the situation. For this purpose, notifications are defined that a push server can use to inform subscribers/ receivers when a subscription is (temporarily) suspended, when a suspended subscription is resumed, and when a a subscription is terminated. This way, receivers will be able to rely on a subscription, knowing that they will be informed of any situations in which updates might be missed.

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. Push data streams

There are several conceptual data streams introduced in this specification:

- yang-push includes the entirety of YANG data, including both configuration and operational data.

- operational-push includes all operational (read-only) YANG data

- config-push includes all YANG configuration data.

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

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

- other custom datastreams

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 yang-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 update-streams
```

```
   |  +--ro update-stream*   update-stream
   +--rw filters
   |  +--rw filter* [filter-id]
   |     +--rw filter-id         filter-id
   |     +--rw (filter-type)?
   |        +--:(subtree)
   |        |  +--rw subtree-filter
   |        +--:(xpath)
   |        |  +--rw xpath-filter?     yang:xpath1.0
   |        +--:(rfc5277)
   |           +--rw filter
   +--rw subscription-config {configured-subscriptions}?
   |  +--rw datastore-push-subscription* [subscription-id]
   |     +--rw subscription-id            subscription-id
   |     +--rw stream?                    update-stream
   |     +--rw encoding?                  encoding
   |     +--rw subscription-start-time?   yang:date-and-time
   |     +--rw subscription-stop-time?    yang:date-and-time
   |     +--rw (filterspec)?
   |     |  +--:(inline)
   |     |  |  +--rw (filter-type)?
   |     |  |     +--:(subtree)
   |     |  |     |  +--rw subtree-filter
   |     |  |     +--:(xpath)
   |     |  |     |  +--rw xpath-filter?              yang:xpath1.0
   |     |  |     +--:(rfc5277)
   |     |  |        +--rw filter
   |     |  +--:(by-reference)
   |     |     +--rw filter-ref?               filter-ref
   |     +--rw (update-trigger)?
   |     |  +--:(periodic)
   |     |  |  +--rw period                    yang:timeticks
   |     |  +--:(on-change) {on-change}?
   |     |     +--rw no-synch-on-start?        empty
   |     |     +--rw dampening-period          yang:timeticks
   |     |     +--rw excluded-change*          change-type
   |     +--rw receiver* [address]
   |     |  +--rw address     inet:host
   |     |  +--rw port?       inet:port-number
   |     |  +--rw protocol?   transport-protocol
   |     +--rw (push-source)?
   |        +--:(interface-originated)
   |        |  +--rw source-interface?         if:interface-ref
   |        +--:(address-originated)
   |           +--rw source-vrf?              uint32
   |           +--rw source-address           inet:ip-address-no-zone
   +--ro subscriptions
      +--ro datastore-push-subscription* [subscription-id]
```

```
      +--ro subscription-id           subscription-id
      +--ro configured-subscription?  empty {configured-subscriptions}?
      +--ro subscription-status?      identityref
      +--ro stream?                   update-stream
      +--ro encoding?                 encoding
      +--ro subscription-start-time?  yang:date-and-time
      +--ro subscription-stop-time?   yang:date-and-time
      +--ro (filterspec)?
      |  +--:(inline)
      |  |  +--ro (filter-type)?
      |  |     +--:(subtree)
      |  |     |  +--ro subtree-filter
      |  |     +--:(xpath)
      |  |     |  +--ro xpath-filter?            yang:xpath1.0
      |  |     +--:(rfc5277)
      |  |        +--ro filter
      |  +--:(by-reference)
      |     +--ro filter-ref?              filter-ref
      +--ro (update-trigger)?
      |  +--:(periodic)
      |  |  +--ro period                   yang:timeticks
      |  +--:(on-change) {on-change}?
      |     +--ro no-synch-on-start?       empty
      |     +--ro dampening-period         yang:timeticks
      |     +--ro excluded-change*         change-type
      +--ro receiver* [address]
      |  +--ro address     inet:host
      |  +--ro port?       inet:port-number
      |  +--ro protocol?   transport-protocol
      +--ro (push-source)?
         +--:(interface-originated)
         |  +--ro source-interface?        if:interface-ref
         +--:(address-originated)
            +--ro source-vrf?              uint32
            +--ro source-address           inet:ip-address-no-zone
```

Figure 9: Model structure

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

4.2. Update streams

Container "update-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.

One of three 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. Finally, filters can be specified using syntax and semantics of RFC5277.

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

4.4. Subscription configuration

As an optional feature, configured-subscriptions, allows for the static configuration of subscriptions, i.e. for subscriptions that are created via configuration as opposed to RPC. Subscriptions configurations are represented by list subscription-config. Each subscription is represented through its own list element and includes the following components:

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

- "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 if the json-enconding feature is supported. Other encodings may be supported in the future.

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

- "subscription-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.

- 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.

- 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.

- This is followed with a list of receivers for the subscription, indicating for each receiver the transport that should be used for push updates (if options other than Netconf are supported). It

should be noted that the receiver does not have to be the same system that configures the subscription.

- Finally, "push-source" can be used to specify the source of push updates, either a specific interface or server address.

A subscription created through configuration cannot be deleted using an RPC. Likewise, subscriptions created through RPC cannot be deleted through configuration.

The deletion of a subscription, whether through RPC or configuration, results in immediate termination of the subsciption.

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 the state of all subscriptions that are currently active. This includes subscriptions that were created (and have not yet been deleted) using RPCs, as well as subscriptions that have been configured as part of configuration.

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. Leaf "configured-subscription" indicates whether the subscription came into being via configuration or via RPC.

Subscriptions that were created by RPC are removed from the list once they expire (reaching stop-time)or when they are terminated. Subscriptions that were created by configuration need to be deleted from the configuration by a configuration editing operation.

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:yang:ietf-yang-push:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/sample-data/1.0"
            select="/ex:foo"/>
      <period>500</period>
      <encoding>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">
    <subscription-result
          xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
       ok
    </subscription-result>
    <subscription-id xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
       52
    </subscription-id>
</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. .

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">
    <subscription-result
          xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
       error-data-not-authorized
    </subscription-result>
</rpc-reply>
```

Figure 12: Create-subscription access denied response

If a request is rejected because the server is not able to serve it, the server SHOULD include in the returned error what subscription parameters would have been accepted for the request. 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:yang:ietf-yang-push:1.0">
      <stream>push-update</stream>
      <filter netconf:type="xpath"
            xmlns:ex="http://example.com/sample-data/1.0"
            select="/ex:foo"/>
      <dampening-period>10</dampening-period>
      <encoding>encode-xml</encoding>
   </create-subscription>
</netconf:rpc>
```

Figure 13: Create-subscription request example 2

A server that cannot serve on-change updates but periodic updates might return the following:

```
<rpc-reply message-id="101"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <subscription-result
          xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
       error-no-such-option
    </subscription-result>
    <period>100</period>
</rpc-reply>
```

Figure 14: Create-subscription error response example 2

4.7.2. Modify-subscription RPC

The subscriber may send a modify-subscription PRC for a subscription previously created using RPC The subscriber may change any subscription parameters by including the new values in the modify-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="102"
      xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modify-subscription
           xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
         <stream>push-update</stream>
         <subscription-id>
            1011
         </subscription-id>
         <filter netconf:type="xpath"
              xmlns:ex="http://example.com/sample-data/1.0"
              select="/ex:foo"/>
         <period>250</period>
         <encoding>encode-xml</encoding>
      </modify-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="102"
   xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <subscription-result
         xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
      ok
   </subscription-result>
   <subscription-id xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
      1011
   </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.

A configured subscription cannot be modified using modify-subscription RPC. Instead, the configuration needs to be edited as needed.

4.7.3. Delete-subscription RPC

To stop receiving updates from a subscription and effectively delete a subscription that had previously been created using a create-subscription RPC, a subscriber 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:yang:ietf-yang-push:1.0">
      <subscription-id>
         1011
      </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

Configured subscriptions cannot be deleted via RPC, but have to be removed from the configuration.

5. YANG module

```
<CODE BEGINS>
file "ietf-yang-push@2016-02-23.yang"

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

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

  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:   Eric Voit
               <mailto:evoit@cisco.com>

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

     Editor:   Ambika Prasad Tripathy
               <mailto:ambtripa@cisco.com>

     Editor:   Einar Nilsen-Nygaard
               <mailto:einarnn@cisco.com>";


  description
    "This module contains conceptual YANG specifications
     for YANG push.";

  revision 2016-02-23 {
    description
      "Changes to grouping structure, RPC definitions, filter
       definitions, origin interface and receiver definitions.";
    reference "YANG Datastore Push, draft-ietf-netconf-yang-push-01";
  }

  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.";
  }
```

```
  feature configured-subscriptions {
    description
      "This feature indicates that management plane configuration
       of subscription is supported.";
  }

  identity subscription-result {
    description
      "Base identity for RPC responses to requests surrounding
       management (e.g. creation, modification) of
       subscriptions.";
  }

  identity ok {
    base subscription-result;
    description
      "OK - RPC was successful and was performed as requested.";
  }

  identity error {
    base subscription-result;
    description
      "RPC was not successful.
       Base identity for error return codes.";
  }

  identity error-no-such-subscription {
    base error;
    description
      "A subscription with the requested subscription ID
       does not exist.";
  }

  identity error-no-such-option {
    base error;
    description
      "A requested parameter setting is not supported.";
  }

  identity error-insufficient-resources {
    base error;
    description
      "The server has insufficient resources to support the
       subscription as requested.";
  }

  identity error-configured-subscription {
    base error;
```

```
    description
      "Cannot apply RPC to a configured subscription, i.e.
       to a subscription that was not created via RPC.";
  }

  identity error-data-not-authorized {
    base error;
    description
      "No read authorization for a requested data node.";
  }

  identity error-other {
    base error;
    description
      "An unspecified error has occurred (catch all).";
  }

  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 is currently unable to provide
       the negotiated updates.";
  }

  identity subscription-errors {
    description
      "Base identity for subscription error status.
       This identity is not to be confused with error return
```

```
       codes for RPCs";
  }

  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 event-stream {
    description
      "Base identity to represent a generic stream of event
       notifications.";
  }

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

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

```
  identity operational-push {
    base update-stream;
    description
      "A conceptual datastream consisting of updates of all
       operational data.";
  }

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

  identity custom-stream {
    base update-stream;
    description
      "A category of customizable datastream for datastore
       updates with contents that have defined by a user.";
  }

  identity netconf-stream {
    base event-stream;
    description
      "Default notification stream";
  }

  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 transport {
    description
      "An identity that represents a transport protocol for event updates";
  }
```

```
  identity netconf {
    base transport;
    description
      "Netconf notifications as a transport";
  }

  identity restconf {
    base transport;
    description
      "Restconf notifications as a transport";
  }

  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 subscription-id {
    type uint32;
    description
      "A type for subscription identifiers.";
  }

  typedef filter-id {
    type uint32;
    description
      "A type to identify filters which can be associated with a
       subscription.";
  }

  typedef subscription-result {
    type identityref {
      base subscription-result;
    }
    description
      "The result of a subscription operation";
  }

  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 transport-protocol {
    type identityref {
      base transport;
    }
    description
      "Specifies transport protocol used to send updates to a
       receiver.";
  }

  typedef push-source {
    type enumeration {
      enum "interface-originated" {
        description
          "Pushes will be sent from a specific interface on a
           Publisher";
      }
      enum "address-originated" {
        description
          "Pushes will be sent from a specific address on a
           Publisher";
      }
    }
    description
```

```
      "Specifies from where objects will be sourced when being pushed
       off a publisher.";
  }

  typedef update-stream {
    type identityref {
      base update-stream;
    }
    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.";
        anyxml subtree-filter {
          description
            "Subtree-filter used to specify the data nodes targeted
             for subscription within a subtree, or subtrees, of a
             conceptual YANG datastore.
             It may include additional criteria,
             allowing users to receive only updates of a limited
             set of data nodes that match those filter criteria.
             This will be used to define what
             updates to include in a stream of update events, i.e.
             to specify for which data nodes update events should be
             generated and specific match expressions that objects
             need to meet. The syntax follows the subtree filter
             syntax specified in RFC 6241, section 6.";
          reference "RFC 6241 section 6";
        }
      }
```

```
      case xpath {
        description
          "XPath filter";
        leaf xpath-filter {
          type yang:xpath1.0;
          description
            "Xpath defining the data items of interest.";
        }
      }
      case rfc5277 {
        anyxml filter {
          description
            "Subtree filter per RFC 5277";
        }
      }
    }
  }

  grouping update-policy {
    description
      "This grouping describes the conditions under which an
       update will be sent as part of an update stream.";
    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;
          mandatory true;
          description
            "Duraton of time which should occur between periodic
             push updates.  Where the anchor of a start-time is
             available, the push will include the objects and their
             values which exist at an exact multiple of timeticks
             aligning to this start-time anchor.";
        }
      }
      case on-change {
        if-feature "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.";
        }
      }
    }
  }

  grouping subscription-info {
    description
      "This grouping describes basic information concerning a
       subscription.";
    leaf stream {
      type update-stream;
      description
        "The stream being subscribed to.";
    }
    leaf encoding {
      type encoding;
      default "encode-xml";
```

```
      description
        "The type of encoding for the subscribed data.
         Default is XML";
    }
    leaf subscription-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 subscription-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. No final push is required unless there
         is exact alignment with the end of a periodic subscription
         period.";
    }
    choice filterspec {
      description
        "The filter to be applied to the stream as part of the
         subscription.  The filter defines which updates of the
         data stream are of interest to a subscriber.
         The filter can be specified in-line
         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 filter which is associated with the
```

```
            subscription.";
        }
      }
    }
  }

  grouping push-source-info {
    description
      "Defines the sender source from which push updates
       for a configured subscription are pushed.";
    choice push-source {
      description
        "Identifies the egress interface on the Publisher from
         which pushed updates will or are being sent.";
      case interface-originated {
        description
          "When the push source is out of an interface on the
           Publisher established via static configuration.";
        leaf source-interface {
          type if:interface-ref;
          description
            "References the interface for pushed updates.";
        }
      }
      case address-originated {
        description
          "When the push source is out of an IP address on the
           Publisher established via static configuration.";
        leaf source-vrf {
          type uint32 {
            range "16..1048574";
          }
          description
            "Label of the vrf.";
        }
        leaf source-address {
          type inet:ip-address-no-zone;
          mandatory true;
          description
            "The source address for the pushed objects.";
        }
      }
    }
  }

  grouping receiver-info {
    description
      "Defines where and how to deliver push updates for a
```

```
       configured subscription.  This includes
       specifying the receiver, as well as defining
       any network and transport aspects when pushing of
       updates occurs outside of Netconf or Restconf.";
    list receiver {
      key "address";
      description
        "A single host or multipoint address intended as a target
         for the pushed updates for a subscription.";
      leaf address {
        type inet:host;
        description
          "Specifies the address for the traffic to reach a
           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 to use for messages
           destined for a receiver.";
      }
      leaf protocol {
        type transport-protocol;
        default "netconf";
        description
          "This leaf specifies the transport protocol used
           to deliver messages destined for the receiver.";
      }
    }
  }

  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.
       In case an error (as indicated by subscription-result)
       is returned, the subscription is
       not created.  In that case, the RPC output
       MAY include suggested parameter settings
       that would have a high likelihood of succeeding in a
       subsequent create-subscription request.";
    input {
      uses subscription-info;
```

```
      uses update-policy;
    }
    output {
      leaf subscription-result {
        type subscription-result;
        mandatory true;
        description
          "Indicates whether subscription is operational,
           or if a problem was encountered.";
      }
      choice result {
        description
          "Depending on the subscription result, different
           data is returned.";
        case success {
          description
            "This case is used when the subscription request
             was successful and a subscription was created as
             a result";
          leaf subscription-id {
            type subscription-id;
            mandatory true;
            description
              "Identifier used for this subscription.";
          }
        }
        case no-success {
          description
            "This case applies when a subscription request
             was not successful and no subscription was
             created as a result.  In this case,
             information MAY be returned that indicates
             suggested parameter settings that would have a
             high likelihood of succeeding in a subsequent
             create-subscription request.";
          uses subscription-info;
          uses update-policy;
        }
      }
    }
  }
  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.
       In case an error is returned (as indicated by
       subscription-result), the subscription is
       not modified and the original subscription parameters
       remain in effect.  In that case, the rpc error response
       MAY include suggested parameter settings
       that would have a high likelihood of succeeding in a
       subsequent modify-subscription request.";
    input {
      leaf subscription-id {
        type subscription-id;
        description
          "Identifier to use for this subscription.";
      }
    }
    output {
      leaf subscription-result {
        type subscription-result;
        mandatory true;
        description
          "Indicates whether subscription was modified
           or if a problem was encountered.
           In case the subscription-result has a value
           other than OK, the original subscription was not
           changed.";
      }
      uses subscription-info;
      uses update-policy;
    }
  }
  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 {
    if-feature "on-change";
    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;
    uses update-policy;
  }
  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;
    uses update-policy;
  }
  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 update-streams {
    config false;
    description
      "This container contains a leaf list of built-in
       streams that are provided by the system.";
    leaf-list update-stream {
      type update-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 {
    if-feature "configured-subscriptions";
    description
      "Contains the list of subscriptions that are configured,
       as opposed to established via RPC or other means.";
    list yang-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 update-policy;
      uses receiver-info;
      uses push-source-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 yang-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 {
        if-feature "configured-subscriptions";
        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 update-policy;
      uses receiver-info;
      uses push-source-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. In addition, the Netconf Authorization Control Model SHOULD be used to control and restrict authorization of subscription configuration.

7. References

7.1. 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>.

7.2. 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-05 (work in progress), February 2016.

[I-D.ietf-netconf-restconf]
Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", I-D draft-ietf-netconf-restconf-09, December 2015.

[I-D.ietf-netconf-yang-patch]
Bierman, A., Bjorklund, M., and K. Watsen, "YANG Patch Media Type", draft-ietf-netconf-yang-patch-07 (work in progress), December 2015.

[I-D.ietf-netmod-yang-json]
Lhotka, L., "JSON Encoding of Data Modeled with YANG", draft-ietf-netmod-yang-json-07 (work in progress), January 2016.

[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

Ambika Prasad Tripathy
Cisco Systems

EMail: ambtripa@cisco.com

Einar Nilsen-Nygaard
Cisco Systems

EMail: einarnn@cisco.com