NETCONF
Internet-Draft
Intended status: Standards Track
Expires: February 21, 2018

A. Clemm
Huawei
E. Voit
Cisco Systems
A. Gonzalez Prieto
VMware
A. Tripathy
E. Nilsen-Nygaard
Cisco Systems
A. Bierman
YumaWorks
B. Lengyel
Ericsson
August 20, 2017

Subscribing to YANG datastore push updates
draft-ietf-netconf-yang-push-08

Abstract

Providing rapid visibility into changes made on YANG configuration and operational objects enables new capabilities such as remote mirroring of configuration and operational state. Via the mechanism described in this document, subscriber applications may request a continuous, customized stream of updates from a YANG datastore.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 21, 2018.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction 3
2. Definitions and Acronyms 4
3. Solution Overview 5
3.1. Event Subscription Model 5
3.2. Negotiation of Subscription Policies 6
3.3. On-Change Considerations 6
3.4. Promise-Theory Considerations 8
3.5. Data Encodings 8
3.6. Datastore filters 9
3.7. Streaming updates 10
3.8. Subscription management 13
3.9. Receiver Authorization 14
3.10. On-change notifiable YANG objects 15
3.11. Other considerations 16
4. A YANG data model for management of datastore push subscriptions 18
4.1. Overview 18
4.2. Subscription configuration 24
4.3. YANG Notifications 25

4.4. YANG RPCs 26
5. YANG module 31
6. IANA Considerations 46
7. Security Considerations 46
8. Acknowledgments 47
9. References 47
9.1. Normative References 47
9.2. Informative References 48
Appendix A. Relationships to other drafts 48
A.1. ietf-netconf-subscribed-notifications 49
A.2. ietf-netconf-netconf-event-notif 49
A.3. ietf-netconf-restconf-notif 49
A.4. voit-notifications2 49
Appendix B. Technologies to be considered for future iterations 50
B.1. Proxy YANG Subscription when the Subscriber and Receiver are different 50
B.2. OpState and Filters 50
B.3. Splitting push updates 51
B.4. Potential Subscription Parameters 52
Appendix C. Changes between revisions 52
Authors' Addresses 54

1. Introduction

Traditional approaches to remote visibility have been built on polling. With polling, data is periodically requested and retrieved by a client from a server to stay up-to-date. However, there are issues associated with polling-based management:

- Polling incurs significant latency. This latency prohibits many application types.

- Polling cycles may be missed, requests may be delayed or get lost, often when the network is under stress and the need for the data is the greatest.

- Polling requests may undergo slight fluctuations, resulting in intervals of different lengths. The resulting data is difficult to calibrate and compare.

- For applications that monitor for changes, many remote polling cycles place ultimately fruitless load on the network, devices, and applications.

A more effective alternative to polling is for an application to receive automatic and continuous updates from a targeted subset of a datastore. Accordingly, there is a need for a service that allows applications to subscribe to updates from a YANG datastore and that

enables the publisher to push and in effect stream those updates. The requirements for such a service have been documented in [RFC7923].

This document defines a corresponding solution that is built on top of "Custom Subscription to Event Notifications" [subscribe]. Supplementing that work are YANG data model augmentations, extended RPCs, and new datastore specific update notifications. Transport options for [subscribe] will work seamlessly with this solution.

2. Definitions and Acronyms

The terms below supplement those defined in [subscribe].

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

Data node update: A data item containing the current value/property of a Data node at the time the data node update was created.

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

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

Notification message: A transport encapsulated update record(s) and/ or event notification(s) intended to be sent to a receiver.

Update notification message: A notification message that contains an update record.

Update record: A representation data node update(s) resulting from the application of a filter for a subscription. An update record will include the value/property of one or more data nodes at a point in time. It may contain the update type for each data node (e.g., add, change, delete). Also included may be metadata/headers such as a subscription-id.

Update trigger: A mechanism that determines when an update record needs to be generated.

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 push update subscription service. This solution supports the dynamic as well as configured subscriptions to information updates from YANG datastores. Subscriptions specify when update notification messages should be sent and what data to include in update records. YANG objects are subsequently pushed from the publisher to the receiver per the terms of the subscription.

3.1. Event Subscription Model

YANG-push subscriptions are defined using a data model that is itself defined in YANG. This model enhances the event subscription model defined in [subscribe] with capabilities that allow subscribers to subscribe to data node updates, specifically to specify the triggers when to generate update records as well as what to include in an update record. Key enhancements include:

- o Specification of selection filters which identify targeted YANG data nodes and/or subtrees within a datastore for which updates are to be pushed.

- o An encoding (using anydata) for the contents of periodic and on-change push updates.

- o Specification of update policies that specify the conditions that trigger the generation and pushing of new update records. There are two types of subscriptions, periodic and on-change.

  * For periodic subscriptions, the trigger is specified by two parameters that define when updates are to be pushed. These parameters are the period interval with which to report updates, and an anchor time which can be used to calculate at which point in time updates need to be assembled and sent.

  * For on-change subscriptions, a trigger occurs whenever a change in the subscribed information is detected. Included are additional parameters such as:

    + Dampening period: In an on-change subscription, the first change that is detected results in an update to be sent immediately. However, sending successive updates whenever further changes are detected might result in quick exhaustion of resources in case of very rapid changes. In order to protect against that, a dampening period is used to specify the interval which must pass before successive update records for the same subscription are generated. The

dampening period collectively applies to the set of all data nodes of a single subscription. This means that on change of an object being subscribed to, an update record containing that object is created either immediately when no dampening period is already in effect, or at the end of a dampening period.

+ Change type: This parameter can be used to reduce the types of datastore changes for which updates are sent (e.g., you might only send when an object is created or deleted, but not when an object value changes).

+ No Synch on start: defines whether or not a complete push-update of all subscribed data will be sent at the beginning of a subscription. Such synchronization establishes the frame of reference for subsequent updates.

3.2. Negotiation of Subscription Policies

A dynamic subscription request SHOULD be declined based on publisher's assessment that it may be unable to provide update records that would meet the terms of the request. However a subscriber may quickly follow up with a new subscription request using different parameters.

Random guessing at different parameters by a subscriber is to be discouraged. Therefore, in order to minimize the number of subscription iterations between subscriber and publisher, dynamic subscriptions SHOULD support a simple negotiation between subscribers and publishers for subscription parameters. This negotiation is in the form of a no-success response to a failed establish or modify subscription request. The no-success message SHOULD include in the returned error response information that, when considered, increases the likelihood of success for subsequent requests. However, there are no guarantees that subsequent requests for this subscriber will in fact be accepted.

Such negotiation information returned from a publisher beyond that from [subscribe] includes hints at acceptable time intervals, size estimates for the number or objects which would be returned from a filter, and the names of targeted objects not found in the publisher's YANG tree.

3.3. On-Change Considerations

On-change subscriptions allow subscribers to subscribe to updates whenever changes to objects occur. As such, on-change subscriptions are particularly effective for data that changes infrequently, yet

that requires applications to be notified whenever a change does occur with minimal delay.

On-change subscriptions tend to be more difficult to implement than periodic subscriptions. Accordingly, on-change subscriptions may not be supported by all implementations or for every object.

Whether or not to accept or reject on-change subscription requests when the scope of the subscription contains objects for which on-change is not supported is up to the server implementation: A server MAY accept an on-change subscription even when the scope of the subscription contains objects for which on-change is not supported. In that case, updates are sent only for those objects within the scope that do support on-change updates whereas other objects are excluded from update records, whether or not their values actually change. In order for a client to determine whether objects support on-change subcriptions, objects are marked accordingly by a server. Accordingly, when subscribing, it is the responsibility of the client to ensure it is aware of which objects support on-change and which do not. For more on how objects are so marked, see Section 3.10.

Alternatively, a server MAY decide to simply reject an on-change subscription in case the scope of the subscription contains objects for which on-change is not supported. In case of a configured subscription, the subsription can be marked as suspended respectively inoperational.

To avoid flooding receivers with repeated updates for subscriptions containing fast-changing objects, or objects with oscillating values, an on-change subscription allows for the definition of a dampening period. Once an update record for a given object is generated, no other updates for this particular subscription will be created until the end of the dampening period. Values sent at the end of the dampening period are the current values of all changed objects which are current at the time the dampening period expires. Changed objects includes those which were deleted or newly created during that dampening period. If an object has returned to its original value (or even has been created and then deleted) during the dampening-period, the last change will still be sent. This will indicate churn is occuring on that object.

In cases where a client wants to have separate dampening periods for different objects, multiple subscriptions with different objects in subscription scope can be 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.

3.4. Promise-Theory Considerations

A subscription to updates from a YANG datastore is intended to obviate the need for polling. However, in order to do so, it is of utmost importance that subscribers can rely on the subscription and have confidence that they will indeed receive the subscribed updates without having to worry updates being silently dropped. In other words, a subscription constitutes a promise on the side of the server to provide the receivers with updates per the terms of the subscription.

Now, there are many reasons why a server may at some point no longer be able to fulfill the terms of the subscription, even if the subscription had been entered into with good faith. For example, the volume of data objects may be larger than anticipated, the interval may prove too short to send full updates in rapid succession, or an internal problem may prevent updates from being collected. If for some reason the server of a subscription is not able to keep its promise, receivers MUST be notified immediately and reliably. The server MUST also update the state of the subscription to indicate that the subscription is in a detrimental state.

A server SHOULD reject a request for a subscription if it is unlikely that the server will be able fulfill the terms of the subscription. In such cases, it is preferable to have a client request another subscription that is less resource intensive (for example, a subscription with longer periodic update intervals), than to subsequently frustrate the receiver with `frequent subscription suspensions.

3.5. Data Encodings

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

3.5.1. 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 [RFC7950]. JSON encoding rules are defined in [RFC7951].

3.5.2. On-Change Subscriptions

In an on-change subscription, updates need to indicate not only values of changed data nodes but also the types of changes that occurred since the last update. Therefore encoding rules for data in

on-change updates will follow YANG-patch operation as specified in [RFC8072]. The YANG-patch will describe what needs to be applied to the earlier state reported by the preceding update, to result in the now-current state. Note that contrary to [RFC8072], objects encapsulated are not restricted to configuration objects only.

3.6. Datastore filters

Subscription policy specifies both the filters and the datastores against which the filters will be applied. The result is the push of information necessary to remotely maintain an extract of publisher's datastore.

Only a single filter can be applied to a subscription at a time. The following selection filter types are included in the yang-push data model, and may be applied against a datastore:

o subtree: A subtree filter identifies one or more subtrees. When specified, updates will only come from the data nodes of selected YANG subtree(s). The syntax and semantics correspond to that specified for [RFC6241] section 6.

o xpath: An xpath filter is an XPath expression which may be meaningfully applied to a datastore. It is the results of this expression which will be pushed.

Filters are intended to be used as selectors that define which objects are within the scope of a subscription. Filters are not intended to be used to store objects based on their current value. Doing so would have a number of implications that would result in significant additional complexity. For example, withough extending encodings for on-change subscriptions, a receiver would not be able to distinguish cases in which an object is no longer included in an update because it was deleted, as opposed to its value simply no longer meeting the filter criteria. While it is possible to define extensions in the future that will support filtering based on values, this is not supported in this version of yang-push and a server MAY reject a subscription request that contains a filter for object values.

Xpath itself provides powerful filtering constructs, and care must be used in filter definition. As an example, consider an xpath filter with a boolean result; such a result will not provide an easily interpretable subset of a datastore. Beyond the boolean example, it is quite possible to define an xpath filter where results are easy for an application to mis-interpret. Consider an xpath filter which only passes a datastore object when interface=up. It is up to the

receiver to understand implications of the presence or absence of objects in each update.

It is not expected that implementations will support comprehensive filter syntax and boundless complexity. It will be up to implementations to describe what is viable, but the goal is to provide equivalent capabilities to what is available with a GET. Implementations MUST reject dynamic subscriptions or suspend configured subscriptions if they include filters which are unsupportable on a platform.

3.7. Streaming updates

Contrary to traditional data retrieval requests, datastore subscription enables an unbounded series of update records to be streamed over time. Two generic notifications for update records have been defined for this: "push-update" and "push-change-update".

A push-update notification defines a complete, filtered update of the datastore per the terms of a subscription. This type of notification is used for continuous updates of periodic subscriptions. A push-update notification can also used be for the on-change subscriptions in two cases. First it will be used as the initial push-update if there is a need to synchronize the receiver at the start of a new subscription. It also MAY be sent if the publisher later chooses to resynch an on-change subscription. The push-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.

A push-change-update notification is the most common type of update for on-change subscriptions. The update record in this case contains a data snippet that indicates the set of changes that data nodes have undergone since the last notification of YANG objects. In other words, this indicates which data nodes have been created, deleted, or have had changes to their values. In cases where multiple changes have occurred and the object has not been deleted, the object's most current value is reported. (In other words, for each object, only one change is reported, not its entire history. Doing so would defeat the purpose of the dampening period.)

These new YANG notifications are encoded and placed within notification messages, which are then queued for egress over the specified transport. The following is an example of an XML encoded notification message over NETCONF transport as per [netconf-notif].

```
        <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>2015-03-09T19:14:56.233Z</time-of-update>
             <datastore-contents>
                <foo>
                   <bar>some_string</bar>
                </foo>
             </datastore-contents>
           </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>2015-03-09T19:14:56.233Z</time-of-update>
             <datastore-changes>
               <alpha xmlns="http://example.com/sample-data/1.0" >
                 <beta>1500</beta>
               </alpha>
             </datastore-changes>
           </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">
   &lt;eventTime&gt;2015-03-09T19:14:56Z&lt;/eventTime&gt;
   <push-change-update xmlns=
       "urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
     &lt;subscription-id&gt;89&lt;/subscription-id&gt;
     &lt;time-of-update&gt;2015-03-09T19:14:56.233Z&lt;/time-of-update&gt;
     &lt;datastore-changes&gt;
       {
        "ietf-yang-patch:yang-patch": {
        "patch-id": [
          null
        ],
        "edit": [
          {
              "edit-id": "edit1",
              "operation": "merge",
              "target": "/alpha/beta",
              "value": {
                  "beta": 1500
              }
          }
        ]
       }
     }
     &lt;/datastore-changes&gt;
   &lt;/push-change-update&gt;
&lt;/notification&gt;
```

Figure 3: Push example for on change with JSON

When the beta leaf is deleted, the publisher 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>2015-03-09T19:14:56.233Z</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

3.8. Subscription management

[subscribe] has been enhanced to support YANG datastore subscription negotiation. These enhancements provide information on why a datastore subscription attempt has failed.

A datastore subscription can be rejected for multiple reasons. This includes the lack of read authorization on a requested data node, or the inability of the publisher push update records as frequently as requested. In such cases, no subscription is established. Instead, the subscription-result with the failure reason is returned as part of the RPC response. As part of this response, a set of alternative subscription parameters MAY be returned that would likely have resulted in acceptance of the subscription request. The subscriber may consider these as part of future subscription attempts.

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

Figure 5: Establish-Subscription example

the publisher 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

3.9. Receiver Authorization

A receiver of subscription data MUST only be sent updates for which they have proper authorization. A server MUST ensure that no non-authorized data is included in push updates. To do so, it needs to apply all corresponding checks applicable at the time of a specific pushed update and if necessary silently remove any non-authorized data from subtrees. This enables YANG data pushed based on subscriptions to be authorized equivalently to a regular data retrieval (get) operation.

Alternatively, a server that wants to avoid having to perform filtering of authorized content on each update MAY instead simply reject a subscription request that contains non-authorized data. It MAY subsequently suspend a subscription in case new objects are created during the course of the subscription for which the receiver does not have the necessary authorization, or in case the authorization privileges of a receiver change over the course of the subscription.

The contextual authorization model for data in YANG datastores is the NETCONF Access Control Model [RFC6536bis], Section 3.2.3. However, there are some differences.

One of these clarifications is that datastore selection MUST NOT return continuous errors as part of an on-change subscription. This includes errors such as when there is not read access to every data node specifically named within the filter. Non-authorized data needs to be either simply dropped or, alternatively, the subscription SHOULD be suspended.

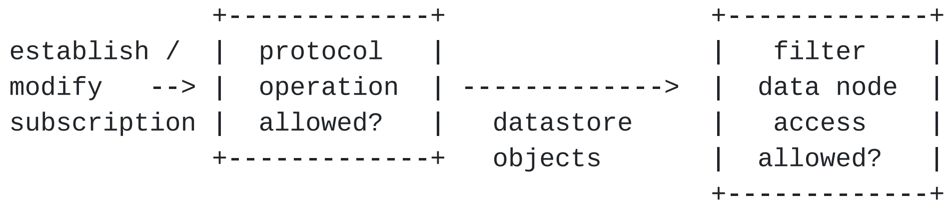

Figure 7: Access control for subscription

Another clarification to [RFC6536bis] is that each of the individual nodes in a resulting update record MUST also have applied access control to resulting pushed messages. This includes validating that read access into new nodes added since the last update record. If read access into previously accessible nodes not explicitly named in the filter are lost mid-subscription, that can be treated as a 'delete' for on-change subscriptions. If not capable of handling such permission changes for dynamic subscriptions, publisher implementations MAY choose to terminate the subscription and to force re-establishment with appropriate filtering.

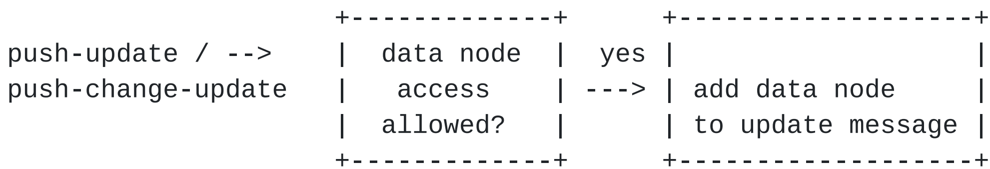

Figure 8: Access control for push updates

3.10. On-change notifiable YANG objects

In some cases, a publisher supporting on-change notifications may not be able to push updates for some object types on-change. Reasons for this might be that the value of the data node changes frequently (e.g., [RFC7223]'s in-octets counter), that small object changes are frequent and meaningless (e.g., a temperature gauge changing 0.1

degrees), or that the implementation is not capable of on-change notification for a particular object.

Support for on-change notification is usually specific to the individual YANG model and/or implementation so it is possible to define in design time. System integrators need this information (without reading any data from a live node).

The default assumption is that no data nodes support on-change notification. Schema nodes and subtrees that support on-change notifications MUST be marked by accordingly with the YANG extension "notifiable-on-change". This extension is defined in the data model below.

When an on-change subscription is established, data-nodes are automatically excluded unless they are marked with notifiable-on-change as true. This also means that authorization checks SHALL NOT be performed on them. A client can identify which nodes will be included in on-change updated by retrieving the data nodes in the subscription's scope and checking for which notifiable-on-change is marked as true.

Adding notifiable-on-change markings will in general require updating the corresponding YANG models. A simple way to avoid having to modify existing module definitions is to add notifiable-on-change markings by defining module deviations. This means that when a YANG model designer wants to add a notifiable-on-change marking to nodes of an existing module without modifying the module definitions, a new module is introduced that contains deviation statements which add "notifiable-on-change" statements as appicable.

```
deviation /sys:system/sys:system-time {
  deviate add {
     yp:notifiable-on-change false;
  }
}
```

Figure 9: Deviation Example

3.11. Other considerations

3.11.1. Robustness and reliability

Particularly in the case of on-change push updates, it is important that push updates do not get lost or, in case the loss of an update is unavoidable, that the receiver is notified accordingly.

Update messages for a single subscription MAY NOT be resequenced.

It is conceivable that under certain circumstances, a publisher will recognize that it is unable to include within an update record the full set of objects desired per the terms of a subscription. In this case, the publisher MUST take one or more of the following actions.

- A publisher MUST set the updates-not-sent flag on any update record which is known to be missing information.

- It MAY choose to suspend a subscription as per [subscribe].

- When resuming an on-change subscription, the publisher SHOULD generate a complete patch from the previous update record. If this is not possible and the synch-on-start option is configured, then the full datastore contents MAY be sent instead (effectively replacing the previous contents). If neither of these are possible, then an updates-not-sent flag MUST be included on the next push-change-update.

Note: It is perfectly acceptable to have a series of push-change-updates (and even push updates) serially queued at the transport layer awaiting transmission. It is not required to merge pending update messages. I.e., the dampening period applies to update record creation, not transmission.

3.11.2. Update size and fragmentation

Depending on the subscription, the volume of updates can become quite large. Additionally, based on the platform, it is possible that update records for a single subscription are best sent independently from different line-cards. Therefore, it may not always be practical to send the entire update record in a single chunk. Implementations may therefore choose, at their discretion, to "chunk" update records, breaking one subscription's objects across several update records. In this case the updates-not-sent flag will indicate that no single update record is complete, and it is permissible for multiple updates to come into a receiver for a single periodic interval or on-change dampening period.

Care must be taken in chunking as problems may arrise for objects that have containment or referential dependencies. The publisher must consider these issues if chunking is provided.

3.11.3. Publisher capacity

It is far preferable to decline a subscription request than to accept such a request when it cannot be met.

Whether or not a subscription can be supported will be determined by a combination of several factors such as the subscription policy (on-change or periodic), 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.

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. Following YANG tree convention in the depiction, brackets enclose list keys, "rw" means configuration, "ro" operational state data, "?" designates optional nodes, "*" designates nodes that can have multiple instances. Parentheses with a name in the middle enclose choice and case nodes. New YANG objects defined here (i.e., beyond those from [subscribe]) are identified with "yp".

```
module: ietf-subscribed-notifications
    +--ro streams
    |  +--ro stream* [stream]
    |  |  +--ro stream                      stream
    |  |  +--ro description                 string
    |  |  +--ro replay-support?             empty
    |  |  +--ro replay-log-creation-time?   yang:date-and-time
    |  |  +--ro replay-log-aged-time?       yang:date-and-time
    |  +--ro yp:datastores
    |     +--ro yp:datastore*   datastore
    +--rw filters
    |  +--rw filter* [identifier]
    |     +--rw identifier           filter-id
    |     +--rw (filter-type)?
    |        +--:(event-filter)
    |           +--rw event-filter-type    event-filter-type
    |           +--rw event-filter         <anyxml>
    +--rw subscription-config {configured-subscriptions}?
    |  +--rw subscription* [identifier]
    |     +--rw identifier                  subscription-id
    |     +--rw encoding?                   encoding
    |     +--rw (target)
    |     |  +--:(event-stream)
    |     |  |  +--rw stream                      stream
    |     |  +--:(yp:datastore)
    |     |     +--rw yp:datastore                datastore
    |     |     +--rw yp:selection-filter-type    selection-filter-type
    |     |     +--rw yp:selection-filter         <anyxml>
```

```
     |     +--rw (applied-filter)
     |     |  +--:(by-reference)
     |     |  |  +--rw filter-ref                  filter-ref
     |     |  +--:(within-subscription)
     |     |     +--rw (filter-type)?
     |     |        +--:(event-filter)
     |     |           +--rw event-filter-type     event-filter-type
     |     |           +--rw event-filter          <anyxml>
     |     +--rw stop-time?                  yang:date-and-time
     |     +--rw receivers
     |     |  +--rw receiver* [address port]
     |     |     +--rw address     inet:host
     |     |     +--rw port        inet:port-number
     |     |     +--rw protocol?   transport-protocol
     |     +--rw (notification-origin)?
     |     |  +--:(interface-originated)
     |     |  |  +--rw source-interface?           if:interface-ref
     |     |  +--:(address-originated)
     |     |     +--rw source-vrf?           string
     |     |     +--rw source-address        inet:ip-address-no-zone
     |     +--rw (yp:update-trigger)?
     |     |  +--:(yp:periodic)
     |     |  |  +--rw yp:period                   yang:timeticks
     |     |  |  +--rw yp:anchor-time?             yang:date-and-time
     |     |  +--:(yp:on-change) {on-change}?
     |     |     +--rw yp:dampening-period         yang:timeticks
     |     |     +--rw yp:no-synch-on-start?       empty
     |     |     +--rw yp:excluded-change*         change-type
     |     +--rw yp:dscp?                    inet:dscp
     |     +--rw yp:weighting?               uint8
     |     +--rw yp:dependency?              sn:subscription-id
     +--ro subscriptions
        +--ro subscription* [identifier]
           +--ro identifier                  subscription-id
           +--ro configured-subscription?    empty
           |                                 {configured-subscriptions}?
           +--ro encoding?                   encoding
           +--ro (target)
           |  +--:(event-stream)
           |  |  +--ro stream                      stream
           |  |  +--ro replay-start-time?          yang:date-and-time
           |  |                                    {replay}?
           |  +--:(yp:datastore)
           |     +--ro yp:datastore                datastore
           |     +--ro yp:selection-filter-type    selection-filter-type
           |     +--ro yp:selection-filter         <anyxml>
           +--ro (applied-filter)
           |  +--:(by-reference)
```

```
           |  |  +--ro filter-ref                  filter-ref
           |  +--:(within-subscription)
           |     +--ro (filter-type)?
           |        +--:(event-filter)
           |           +--ro event-filter-type     event-filter-type
           |           +--ro event-filter          <anyxml>
           +--ro stop-time?                  yang:date-and-time
           +--ro (notification-origin)?
           |  +--:(interface-originated)
           |  |  +--ro source-interface?     if:interface-ref
           |  +--:(address-originated)
           |     +--ro source-vrf?           string
           |     +--ro source-address        inet:ip-address-no-zone
           +--ro receivers
           |  +--ro receiver* [address port]
           |     +--ro address                   inet:host
           |     +--ro port                      inet:port-number
           |     +--ro protocol?                 transport-protocol
           |     +--ro pushed-notifications?     yang:counter64
           |     +--ro excluded-notifications?   yang:counter64
           |     +--ro status                    subscription-status
           +--ro (yp:update-trigger)?
           |  +--:(yp:periodic)
           |  |  +--ro yp:period                   yang:timeticks
           |  |  +--ro yp:anchor-time?             yang:date-and-time
           |  +--:(yp:on-change) {on-change}?
           |     +--ro yp:dampening-period         yang:timeticks
           |     +--ro yp:no-synch-on-start?       empty
           |     +--ro yp:excluded-change*         change-type
           +--ro yp:dscp?                    inet:dscp
           +--ro yp:weighting?               uint8
           +--ro yp:dependency?              sn:subscription-id

   rpcs:
     +---x establish-subscription
     |  +---w input
     |  |  +---w encoding?                   encoding
     |  |  +---w (target)
     |  |  |  +--:(event-stream)
     |  |  |  |  +---w stream                      stream
     |  |  |  |  +---w replay-start-time?          yang:date-and-time
     |  |  |  |                                    {replay}?
     |  |  |  +--:(yp:datastore)
     |  |  |     +---w yp:datastore                datastore
     |  |  |     +---w yp:selection-filter-type    selection-filter-type
     |  |  |     +---w yp:selection-filter         <anyxml>
     |  |  +---w (applied-filter)
     |  |  |  +--:(by-reference)
```

```
     |  |  |  |  +---w filter-ref                  filter-ref
     |  |  |  +--:(within-subscription)
     |  |  |     +---w (filter-type)?
     |  |  |        +--:(event-filter)
     |  |  |           +---w event-filter-type     event-filter-type
     |  |  |           +---w event-filter          <anyxml>
     |  |  +---w stop-time?                  yang:date-and-time
     |  |  +---w (yp:update-trigger)?
     |  |  |  +--:(yp:periodic)
     |  |  |  |  +---w yp:period                   yang:timeticks
     |  |  |  |  +---w yp:anchor-time?             yang:date-and-time
     |  |  |  +--:(yp:on-change) {on-change}?
     |  |  |     +---w yp:dampening-period         yang:timeticks
     |  |  |     +---w yp:no-synch-on-start?       empty
     |  |  |     +---w yp:excluded-change*         change-type
     |  |  +---w yp:dscp?                    inet:dscp
     |  |  +---w yp:weighting?               uint8
     |  |  +---w yp:dependency?              sn:subscription-id
     |  +--ro output
     |     +--ro subscription-result         subscription-result
     |     +--ro (result)?
     |        +--:(no-success)
     |        |  +--ro filter-failure?             string
     |        |  +--ro replay-start-time-hint?     yang:date-and-time
     |        |  +--ro yp:period-hint?             yang:timeticks
     |        |  +--ro yp:error-path?              string
     |        |  +--ro yp:object-count-estimate?   uint32
     |        |  +--ro yp:object-count-limit?      uint32
     |        |  +--ro yp:kilobytes-estimate?      uint32
     |        |  +--ro yp:kilobytes-limit?         uint32
     |        +--:(success)
     |           +--ro identifier                  subscription-id
     +---x modify-subscription
     |  +---w input
     |  |  +---w identifier?            subscription-id
     |  |  +---w (applied-filter)
     |  |  |  +--:(by-reference)
     |  |  |  |  +---w filter-ref             filter-ref
     |  |  |  +--:(within-subscription)
     |  |  |     +---w (filter-type)?
     |  |  |        +--:(event-filter)
     |  |  |           +---w event-filter-type      event-filter-type
     |  |  |           +---w event-filter           <anyxml>
     |  |  +---w stop-time?             yang:date-and-time
     |  |  +---w (yp:update-trigger)?
     |  |     +--:(yp:periodic)
     |  |     |  +---w yp:period              yang:timeticks
     |  |     |  +---w yp:anchor-time?        yang:date-and-time
```

```
     |  |     +--:(yp:on-change) {on-change}?
     |  |        +---w yp:dampening-period    yang:timeticks
     |  +--ro output
     |     +--ro subscription-result         subscription-result
     |     +--ro (result)?
     |        +--:(no-success)
     |           +--ro filter-failure?             string
     |           +--ro yp:period-hint?             yang:timeticks
     |           +--ro yp:error-path?              string
     |           +--ro yp:object-count-estimate?   uint32
     |           +--ro yp:object-count-limit?      uint32
     |           +--ro yp:kilobytes-estimate?      uint32
     |           +--ro yp:kilobytes-limit?         uint32
     +---x delete-subscription
     |  +---w input
     |  |  +---w identifier    subscription-id
     |  +--ro output
     |     +--ro subscription-result    subscription-result
     +---x kill-subscription
        +---w input
        |  +---w identifier    subscription-id
        +--ro output
           +--ro subscription-result    subscription-result

   notifications:
     +---n replay-complete
     |  +--ro identifier    subscription-id
     +---n notification-complete
     |  +--ro identifier    subscription-id
     +---n subscription-started
     |  +--ro identifier                  subscription-id
     |  +--ro encoding?                   encoding
     |  +--ro (target)
     |  |  +--:(event-stream)
     |  |  |  +--ro stream                      stream
     |  |  |  +--ro replay-start-time?          yang:date-and-time
     |  |  |                                    {replay}?
     |  |  +--:(yp:datastore)
     |  |     +--ro yp:datastore                datastore
     |  |     +--ro yp:selection-filter-type    selection-filter-type
     |  |     +--ro yp:selection-filter         <anyxml>
     |  +--ro (applied-filter)
     |  |  +--:(by-reference)
     |  |  |  +--ro filter-ref                  filter-ref
     |  |  +--:(within-subscription)
     |  |     +--ro (filter-type)?
     |  |        +--:(event-filter)
     |  |           +--ro event-filter-type           event-filter-type
```

```
     |  |           +--ro event-filter                <anyxml>
     |  +--ro stop-time?                   yang:date-and-time
     |  +--ro (yp:update-trigger)?
     |  |  +--:(yp:periodic)
     |  |  |  +--ro yp:period                    yang:timeticks
     |  |  |  +--ro yp:anchor-time?              yang:date-and-time
     |  |  +--:(yp:on-change) {on-change}?
     |  |     +--ro yp:dampening-period          yang:timeticks
     |  |     +--ro yp:no-synch-on-start?        empty
     |  |     +--ro yp:excluded-change*          change-type
     |  +--ro yp:dscp?                     inet:dscp
     |  +--ro yp:weighting?                uint8
     |  +--ro yp:dependency?               sn:subscription-id
     +---n subscription-resumed
     |  +--ro identifier    subscription-id
     +---n subscription-modified
     |  +--ro identifier              subscription-id
     |  +--ro encoding?               encoding
     |  +--ro (target)
     |  |  +--:(event-stream)
     |  |     +--ro stream                  stream
     |  |     +--ro replay-start-time?      yang:date-and-time {replay}?
     |  +--ro (applied-filter)
     |  |  +--:(by-reference)
     |  |  |  +--ro filter-ref              filter-ref
     |  |  +--:(within-subscription)
     |  |     +--ro (filter-type)?
     |  |        +--:(event-filter)
     |  |           +--ro event-filter-type       event-filter-type
     |  |           +--ro event-filter            <anyxml>
     |  +--ro stop-time?              yang:date-and-time
     |  +--ro (yp:update-trigger)?
     |  |  +--:(yp:periodic)
     |  |  |  +--ro yp:period               yang:timeticks
     |  |  |  +--ro yp:anchor-time?         yang:date-and-time
     |  |  +--:(yp:on-change) {on-change}?
     |  |     +--ro yp:dampening-period     yang:timeticks
     |  |     +--ro yp:no-synch-on-start?   empty
     |  |     +--ro yp:excluded-change*     change-type
     |  +--ro yp:dscp?                inet:dscp
     |  +--ro yp:weighting?           uint8
     |  +--ro yp:dependency?          sn:subscription-id
     +---n subscription-terminated
     |  +--ro identifier        subscription-id
     |  +--ro error-id          subscription-errors
     |  +--ro filter-failure?   string
     +---n subscription-suspended
        +--ro identifier        subscription-id
```

```
        +--ro error-id          subscription-errors
        +--ro filter-failure?   string

 module: ietf-yang-push

   notifications:
     +---n push-update
     |  +--ro subscription-id       sn:subscription-id
     |  +--ro time-of-update?       yang:date-and-time
     |  +--ro updates-not-sent?     empty
     |  +--ro datastore-contents?   <anydata>
     +---n push-change-update {on-change}?
        +--ro subscription-id      sn:subscription-id
        +--ro time-of-update?      yang:date-and-time
        +--ro updates-not-sent?    empty
        +--ro datastore-changes?   <anydata>
```

Figure 10: Model structure

Selected components of the model are summarized below.

4.2. Subscription configuration

Both configured and dynamic subscriptions are represented within the list subscription-config. Each subscription has own list elements. New and enhanced parameters extending the basic subscription data model in [subscribe] include:

- An update filter identifying yang nodes of interest. Filter contents are specified via a reference to an existing filter, or via an in-line definition for only that subscription. This facilitates the reuse of filter definitions, which can be important in case of complex filter conditions. Referenced filters can also allow an implementation to avoid evaluating filter acceptability during a dynamic subscription request. The case statement differentiates the options.

- For periodic subscriptions, triggered updates will occur at the boundaries of a specified time interval. These boundaries many be calculated from the periodic parameters:

  * a "period" which defines duration between period push updates.

  * an "anchor-time"; update intervals always fall on the points in time that are a multiple of a period after the anchor time. If anchor time is not provided, then the anchor time MUST be set with the creation time of the initial update record.

o For on-change subscriptions, assuming the dampening period has completed, triggered occurs whenever a change in the subscribed information is detected. On-change subscriptions have more complex semantics that is guided by its own set of parameters:

 * a "dampening-period" specifies the interval that must pass before a successive update for the subscription is sent. If no dampening period is in effect, the update is sent immediately. If a subsequent change is detected, another update is only sent once the dampening period has passed for this subscription.

 * an "excluded-change" flag which allows restriction of the types of changes for which updates should be sent (e.g., only add to an update record on object creation).

 * a "no-synch-on-start" flag which specifies whether a complete update with all the subscribed data is to be sent at the beginning of a subscription.

o Optional qos parameters to indicate the treatment of a subscription relative to other traffic between publisher and receiver. These include:

 * A "dscp" QoS marking which MUST be stamped on notification messages to differentiate network QoS behavior.

 * A "weighting" so that bandwidth proportional to this weighting can be allocated to this subscription relative to others for that receiver.

 * a "dependency" upon another subscription. Notification messages MUST NOT be sent prior to other notification messages containing update record(s) for the referenced subscription.

o A subscription's weighting MUST work identically to stream dependency weighting as described within RFC 7540, section 5.3.2.

o A subscription's dependency MUST work identically to stream dependency as described within RFC 7540, sections 5.3.1, 5.3.3, and 5.3.4. If a dependency is attempted via an RPC, but the referenced subscription does not exist, the dependency will be removed.

4.3. YANG Notifications

4.3.1. Monitoring and OAM Notifications

OAM notifications and mechanism are reused from [subscribe]. Some have been augmented to include the YANG datastore specific objects.

4.3.2. New Notifications for update records

The data model introduces two YANG notifications to encode information for update records: "push-update" and "push-change-update".

"Push-update" is used to send a complete snapshot of the filtered subscription data. This type of notification is used to carry the update records of a periodic subscription. The "push-update" notification is also used with on-change subscriptions for the purposes of allowing a receiver to "synch" on a complete set of subscribed datastore contents. This synching may be done the start of an on-change subscription, and then later in that subscription to force resynchronization. If the "updates-not-sent" flag is set, this indicates that the update record is incomplete.

"Push-change-update" is used to send datastore changes that have occurred in subscribed data since the previous update. This notification is used only in conjunction with on-change subscriptions. This will be encoded as yang-patch data.

If the application detects an informational discontinuity in either notification, the notification MUST include a flag "updates-not-sent". This flag which indicates that not all changes which have occurred since the last update are actually included with this update. In other words, the publisher has failed to fulfill its full subscription obligations. (For example a datastore missed a window in providing objects to a publisher process.) To facilitate synchronization, a publisher MAY subsequently send a push-update containing a full snapshot of subscribed data.

4.4. YANG RPCs

YANG-Push subscriptions are established, modified, and deleted using RPCs augmented from [subscribe].

4.4.1. Establish-subscription RPC

The subscriber sends an establish-subscription RPC with the parameters in section 3.1. An example might look like:

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

Figure 11: Establish-subscription RPC

The publisher 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 publisher 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 12: Establish-subscription positive RPC response

A subscription can be rejected for multiple reasons, including the lack of authorization to establish a subscription, the lack of read authorization on the requested data node, or the inability of the publisher to provide a stream with the requested semantics.

When the requester is not authorized to read the requested data node, the returned information indicates the node is unavailable. For instance, if the above request was unauthorized to read node "ex:foo" the publisher 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">
       subtree-unavailable
    </subscription-result>
    <filter-failure
          xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
       /ex:foo
    </filter-failure>
</rpc-reply>
```

Figure 13: Establish-subscription access denied response

If a request is rejected because the publisher is not able to serve it, the publisher SHOULD include in the returned error what subscription parameters would have been accepted for the request. However, there are no guarantee that subsequent requests using this info 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">
   <establish-subscription
         xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
      <datastore>running</datastore>
      <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>
   </establish-subscription>
</netconf:rpc>
```

Figure 14: Establish-subscription request example 2

a publisher 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">
       period-unsupported
    </subscription-result>
    <period-hint>100</period-hint>
</rpc-reply>
```

Figure 15: Establish-subscription error response example 2

4.4.2. Modify-subscription RPC

The subscriber MAY invoke the modify-subscription RPC for a subscription it previously established. The subscriber will include newly desired values in the modify-subscription RPC. Parameters not included MUST remain unmodified. Below is an example where a subscriber attempts to modify the period of a 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">
      <datastore>running</datastore>
      <subscription-id>
         1011
      </subscription-id>
      <period>250</period>
   </modify-subscription>
</netconf:rpc>
```

Figure 16: Modify subscription request

The publisher MUST respond explicitly positively or negatively to the request. A response to a successful modification might look like:

```
<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>
</rpc-reply>
```

Figure 17: Modify subscription response

If the subscription modification is rejected, the publisher MUST send a response like it does for an establish-subscription and maintain

the subscription as it was before the modification request. Responses MAY include hints. 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.

4.4.3. Delete-subscription RPC

To stop receiving updates from a subscription and effectively delete a subscription that had previously been established using an establish-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">
   <subscription-result
      xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push:1.0">
         ok
   </subscription-result>
</rpc-reply>
```

Figure 18: Delete subscription

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

4.4.4. YANG Module Synchronization

To make subscription requests, the subscriber needs to know the YANG module library available on the publisher. The YANG 1.0 module library information is sent by a NETCONF server in the NETCONF 'hello' message. For YANG 1.1 modules and all modules used with the RESTCONF [RFC8040] protocol, this information is provided by the YANG Library module (ietf-yang-library.yang from [RFC7895]. The YANG

library information is important for the receiver to reproduce the set of object definitions used by the replicated datastore.

The YANG library includes a module list with the name, revision, enabled features, and applied deviations for each YANG module implemented by the publisher. The receiver is expected to know the YANG library information before starting a subscription. The "/modules-state/module-set-id" leaf in the "ietf-yang-library" module can be used to cache the YANG library information.

The set of modules, revisions, features, and deviations can change at run-time (if supported by the server implementation). In this case, the receiver needs to be informed of module changes before data nodes from changed modules can be processed correctly. The YANG library provides a simple "yang-library-change" notification that informs the client that the library has changed. The receiver then needs to re-read the entire YANG library data for the replicated server in order to detect the specific YANG library changes. The "ietf-netconf-notifications" module defined in [RFC6470] contains a "netconf-capability-change" notification that can identify specific module changes. For example, the module URI capability of a newly loaded module will be listed in the "added-capability" leaf-list, and the module URI capability of an removed module will be listed in the "deleted-capability" leaf-list.

5. YANG module

```
<CODE BEGINS>; file "ietf-yang-push@2017-08-20.yang"
module ietf-yang-push {
  yang-version 1.1;
  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-subscribed-notifications {
    prefix sn;
  }

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

     Editor:   Alexander Clemm
               <mailto:ludwig@clemm.org>

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

     Editor:   Alberto Gonzalez Prieto
               <mailto:agonzalezpri@vmware.com>

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

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

     Editor:   Andy Bierman
               <mailto:andy@yumaworks.com>

     Editor:   Balazs Lengyel
               <mailto:balazs.lengyel@ericsson.com>";

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

  revision 2017-08-20 {
    description
      "Initial revision.";
    reference
      "YANG Datastore Push, draft-ietf-netconf-yang-push-08";
  }

 /*
  * EXTENSIONS
  */

  extension notifiable-on-change {
    argument "value";
    description
      "Indicates whether changes to the data node are reportable in
      on-change subscriptions.
```

```
      The statement MUST only be a substatement of the leaf, leaf-list,
      container, list, anyxml, anydata  statements. Zero or One
      notifiable-on-change statement is allowed per parent statement.
      NO substatements are allowed.

      The argument is a boolean value indicating whether on-change
      notifications are supported. If notifiable-on-change is not
      specified, the default is the same as the parent data node's
      value. For top level data nodes the default value is false.";
  }

 /*
  * FEATURES
  */

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

 /*
  * IDENTITIES
  */

  /* Error type identities for datastore subscription  */
  identity period-unsupported {
    base sn:error;
    description
      "Requested time period is too short. This can be for both
       periodic and on-change dampening.";
  }

  identity qos-unsupported {
    base sn:error;
    description
      "Subscription QoS parameters not supported on this platform.";
  }

  identity dscp-unavailable {
    base sn:error;
    description
      "Requested DSCP marking not allocatable.";
  }

  identity on-change-unsupported {
    base sn:error;
    description
```

```
      "On-change not supported.";
  }

  identity synch-on-start-unsupported {
    base sn:error;
    description
      "On-change synch-on-start not supported.";
  }

  identity synch-on-start-datatree-size {
    base sn:error;
    description
      "Synch-on-start would push a datatree which exceeds size limit.";
  }

  identity reference-mismatch {
    base sn:error;
    description
      "Mismatch in filter key and referenced yang subtree.";
  }

  identity data-unavailable {
    base sn:error;
    description
      "Referenced yang node or subtree doesn't exist, or read
       access is not permitted.";
  }

  identity datatree-size {
    base sn:error;
    description
      "Resulting push updates would exceed size limit.";
  }

  /* Datastore identities */
  identity datastore {
    description
      "Abstract base identity for datastore identities.  More are in the
          process of being defined within
          draft-ietf-netmod-revised-datastores. ";
  }
  identity candidate {
    base datastore;
    description
      "The candidate datastore per RFC-6241.";
    reference "RFC-6241, #5.1";
  }
  identity running {
```

```
    base datastore;
    description
      "The running datastore per RFC-6241.";
    reference "RFC-6241, #5.1";
  }
  identity startup {
    base datastore;
    description
      "The startup datastore per RFC-6241.";
    reference "RFC-6241, #5.1";
  }
  identity operational {
    base datastore;
    description
      "The operational datastore contains all configuration data
      actually used by the system, including all applied configuration,
      system-provided configuration and values defined by any supported
      data models.  In addition, the operational datastore also
      contains state data.";
    reference
      "the original text came from draft-ietf-netmod-revised-datastores
      -01, section #4.3. This definition is expected to remain stable
      meaning later reconciliation between the drafts unnecessary.";
  }
  identity intended {
    base datastore;
    description
      "The intended datastore per draft-ietf-netmod-revised-datastores.";
    reference
      "draft-ietf-netmod-revised-datastores";
  }
  identity custom-datastore {
    base datastore;
    description
      "Another datastore not defined via an identity in this model";
  }

  /*  Selection filter identities */
  identity selection-filter {
    description
      "Evaluation criteria encoded in a syntax which allows the
          selection of nodes from a target.  If the filter is applied
          against a datastore for periodic extracts, the resulting node-set
          result is passed along. If the filter is applied against a
          datastore looking for changes, deltas from the last update in the
          form of a patch result are passed along.  An empty node set is a
          valid result of this filter type.";
  }
```

```
  identity subtree-selection-filter {
    base selection-filter;
    description
      "An RFC-6241 based selection-filter which may be used to select
          nodes within a datastore.";
    reference "RFC-6241, #5.1";
  }
  identity xpath-selection-filter {
    base selection-filter;
    description
      "A selection-filter which may be applied to a datastore which
          follows the syntax specified in yang:xpath1.0.  Nodes that evaluate
      to true are included in the selection.";
    reference "XPATH: http://www.w3.org/TR/1999/REC-xpath-19991116";
  }

  /*
   * TYPE DEFINITIONS
   */

  typedef change-type {
    type enumeration {
      enum "create" {
        description
          "Create a new data resource if it does not already exist.  If
          it already exists, replace.";
      }
      enum "delete" {
        description
          "Delete a data resource if it already exists. If it does not
          exists, take no action.";
      }
      enum "insert" {
        description
          "Insert a new user-ordered data resource";
      }
      enum "merge" {
        description
          "merge the edit value with the target data resource; create
          if it does not already exist";
      }
      enum "move" {
        description
          "Reorder the target data resource";
      }
      enum "replace" {
        description
          "Replace the target data resource with the edit value";
```

```
      }
      enum "remove" {
        description
          "Remove a data resource if it already exists ";
      }
    }
    description
      "Specifies different types of datastore changes.";
    reference
      "RFC 8072 section 2.5, with a delta that it is ok to receive
      ability create on an existing node, or recieve a delete on a
      missing node.";
  }

  typedef datastore {
    type identityref {
      base datastore;
    }
    description
      "Specifies a system-provided datastore.  May also specify ability
      portion of a datastore, so as to reduce the filtering effort.";
  }

  typedef selection-filter-type {
    type identityref {
      base selection-filter;
    }
    description
      "Specifies a known type of selection filter.";
  }


  /*
   * GROUP DEFINITIONS
   */

  grouping datastore-criteria {
    description
      "A grouping to define criteria for which objects from which
       datastore to include in push updates.";
    leaf datastore {
      type datastore;
      mandatory true;
      description
          "Datastore against which the subscription has been
           applied.";
    }
    uses selection-filter-objects;
```

```
  }

  grouping selection-filter-objects {
    description
      "This grouping defines a selector for objects from a
       datastore.";
    leaf selection-filter-type {
      type selection-filter-type;
      mandatory true;
      description
        "A filter needs to be a known and understood syntax if it is
        to be interpretable by a device.";
        }
    anyxml selection-filter {
      mandatory true;
      description
        "Datastore evaluation criteria encoded in a syntax of a
                supported type of selection filter.";
    }
  }

  grouping update-policy-modifiable {
    description
      "This grouping describes the datastore specific subscription
       conditions that can be changed during the lifetime of the
       subscription.";
    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 as defined by the selection filter.";
        leaf period {
          type yang:timeticks;
          mandatory true;
          description
            "Duration 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.";
        }
        leaf anchor-time {
          type yang:date-and-time;
          description
            "Designates a timestamp from which the series of periodic
```

```
             push updates are 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.";
        }
      }
      case on-change {
        if-feature "on-change";
        description
          "The agent is requested to notify changes in values in the
           datastore subset as defined by a selection filter.";
        leaf dampening-period {
          type yang:timeticks;
          mandatory true;
          description
            "The shortest time duration which is allowed between the
                        creation of independent yang object update messages.
                        Effectively this is the amount of time that needs to
have
                        passed since the last update.";
        }
      }
    }
  }

  grouping update-policy {
    description
      "This grouping describes the datastore specific subscription
       conditions of a subscription.";
    uses update-policy-modifiable {
      augment "update-trigger/on-change" {
        description
          "Includes objects not modifiable once subscription is
           established.";
        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 only are sent unless the

```
               Publisher chooses to resynch the subscription again.";
          }
          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 update-qos {
      description
        "This grouping describes Quality of Service information
         concerning a subscription.  This information is passed to lower
         layers for transport prioritization and treatment";
      leaf dscp {
        type inet:dscp;
        default "0";
        description
          "The push update's IP packet transport priority. This is made
           visible across network hops to receiver. The transport
           priority is shared for all receivers of a given subscription.";
      }
      leaf weighting {
        type uint8 {
           range "0 .. 255";
        }
        description
          "Relative weighting for a subscription. Allows an underlying
           transport layer perform informed load balance allocations
           between various subscriptions";
        reference
          "RFC-7540, section 5.3.2";
      }
      leaf dependency {
        type sn:subscription-id;
        description
          "Provides the Subscription ID of a parent subscription which
           has absolute priority should that parent have push updates
           ready to egress the publisher. In other words, there should be
           no streaming of objects from the current subscription if of
           the parent has something ready to push.";
        reference
          "RFC-7540, section 5.3.1";
      }
```

```
  }

  grouping update-error-hints {
    description
      "Allow return additional negotiation hints that apply
       specifically to push updates.";
    leaf period-hint {
      type yang:timeticks;
      description
        "Returned when the requested time period is too short. This
         hint can assert an viable period for both periodic push
         cadence and on-change dampening.";
    }
    leaf error-path {
      type string;
      description
        "Reference to a YANG path which is associated with the error
         being returned.";
    }
    leaf object-count-estimate {
      type uint32;
      description
        "If there are too many objects which could potentially be
         returned by the selection filter, this identifies the estimate
                 of the number of objects which the filter would potentially
                 pass.";
    }
    leaf object-count-limit {
      type uint32;
      description
        "If there are too many objects which could be returned by the
         selection filter, this identifies the upper limit of the
                 publisher's ability to service for this subscription.";
    }
    leaf kilobytes-estimate {
      type uint32;
      description
        "If the returned information could be beyond the capacity of
         the publisher, this would identify the data size which could
         result from this selection filter.";
    }
    leaf kilobytes-limit {
      type uint32;
      description
        "If the returned information would be beyond the capacity of
         the publisher, this identifies the upper limit of the
         publisher's ability to service for this subscription.";
    }
```

```
  }

  /*
   * DATA NODES
   */

  augment "/sn:streams" {
    description
      "This augmentation adds datastores to the list of items that
       can be subscribed.";
      container datastores {
        config false;
        description
          "This container contains a leaf list of built-in
           streams that are provided by the system.";
      leaf-list datastore {
        type datastore;
        description
          "Identifies the built-in datastores that are supported by
           the system.  Built-in datastores are associated with their
           own identities. In case configurable custom datastores are
           supported, as indicated by the custom-datastore identity.
           The configuration and delivery of those custom datastores is
           provided separately.";
      }
    }
  }
  augment "/sn:establish-subscription/sn:input" {
    description
      "This augmentation adds additional subscription parameters that
      apply specifically to datastore updates to RPC input.";
    uses update-policy;
    uses update-qos;
  }
  augment "/sn:establish-subscription/sn:input/sn:target" {
    description
      "This augmentation adds the datastore as a valid parameter object
      for the subscription to RPC input.  This provides a target for
      the filter.";
    case datastore {
       uses datastore-criteria;
    }
  }
  augment "/sn:establish-subscription/sn:output/"+
    "sn:result/sn:no-success" {
    description
      "This augmentation adds datastore specific error info
      and hints to RPC output.";
```

```
    uses update-error-hints;
  }
  augment "/sn:modify-subscription/sn:input" {
    description
      "This augmentation adds additional subscription parameters
       specific to datastore updates.";
    uses update-policy-modifiable;
  }
  augment "/sn:modify-subscription/sn:output/"+
    "sn:result/sn:no-success" {
    description
      "This augmentation adds push datastore error info and hints to
      RPC output.";
    uses update-error-hints;
  }
  notification push-update {
    description
      "This notification contains a push update, containing data
       subscribed to via a subscription. This notification is sent for
       periodic updates, for a periodic subscription.  It can also be
       used for synchronization updates of an on-change subscription.
       This notification shall only be sent to receivers of a
       subscription; it does not constitute a general-purpose
       notification.";
    leaf subscription-id {
      type sn: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.";
    }
    leaf updates-not-sent {
      type empty;
      description
        "This is a flag which indicates that not all data nodes
         subscribed to are included with this update. In other words,
         the publisher has failed to fulfill its full subscription
         obligations. This may lead to intermittent loss of
         synchronization of data at the client.  Synchronization at the
         client can occur when the next push-update is received.";
    }
    anydata datastore-contents {
      description
```

```
        "This contains the updated data.  It constitutes a snapshot
         at the time-of-update of the set of data that has been
         subscribed to.  The format and syntax of the data
         corresponds to the format and syntax of data that would be
         returned in a corresponding get operation with the same
         selection filter parameters applied.";
    }
  }
  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 sn: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.";
    }
    leaf updates-not-sent {
      type empty;
      description
        "This is a flag which indicates that not all changes which
         have occurred since the last update are included with this
         update.  In other words, the publisher has failed to
         fulfill its full subscription obligations, for example in
         cases where it was not able to keep up with a change burst.
         To facilitate synchronization, a publisher may subsequently
         send a push-update containing a full snapshot of subscribed
         data. Such a push-update might also be triggered by a
         subscriber requesting an on-demand synchronization.";
    }
    anydata datastore-changes {
      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 yang-
         patch operation, i.e. a yang-patch operation applied to the
         YANG datastore implied by the previous update to result in the
```

```
          current state (and assuming yang-patch could also be applied
          to operational data).";
      }
    }
    augment "/sn:subscription-started" {
      description
        "This augmentation adds many yang datastore specific objects to
         the notification that a subscription has started.";
      uses update-policy;
      uses update-qos;
    }
    augment "/sn:subscription-started/sn:target" {
      description
        "This augmentation allows the datastore to be included as part
        of the notification that a subscription has started.";
      case datastore {
         uses datastore-criteria;
      }
    }
    augment "/sn:subscription-modified" {
      description
        "This augmentation adds many yang datastore specific objects to
         the notification that a subscription has been modified.";
      uses update-policy;
      uses update-qos;
    }
    augment "/sn:subscription-config/sn:subscription" {
      description
        "This augmentation adds many yang datastore specific objects
        which can be configured as opposed to established via RPC.";
      uses update-policy;
      uses update-qos;
    }
    augment "/sn:subscription-config/sn:subscription/sn:target" {
      description
        "This augmentation adds the datastore to the selection filtering
        criteria for a subscription.";
      case datastore {
         uses datastore-criteria;
      }
    }
    augment "/sn:subscriptions/sn:subscription" {
      yp:notifiable-on-change true;
      description
        "This augmentation adds many datastore specific objects to a
         subscription.";
      uses update-policy;
      uses update-qos;
```

```
  }
  augment "/sn:subscriptions/sn:subscription/sn:target" {
    description
      "This augmentation allows the datastore to be included as part
      of the selection filtering criteria for a subscription.";
    case datastore {
       uses datastore-criteria;
    }
  }
}

<CODE ENDS>
```

6. IANA Considerations

This document registers the following namespace URI in the "IETF XML Registry" [RFC3688]:

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

This document registers the following YANG module in the "YANG Module Names" registry [RFC6020]:

Name: ietf-yang-push
Namespace: urn:ietf:params:xml:ns:yang:ietf-yang-push
Prefix: yp
Reference: draft-ietf-netconf-yang-push-08.txt (RFC form)

7. Security Considerations

All security considerations from [subscribe] are relevant for datastores. In addition there are specific security considerations for receviers defined in Section 3.9

If the access control permissions on subscribed YANG nodes change during the lifecycle of a subscription, a publisher MUST either transparently conform to the new access control permissions, or must terminate or restart the subscriptions so that new access control permissions are re-established.

The NETCONF Authorization Control Model SHOULD be used to restrict the delivery of YANG nodes for which the receiver has no access.

8. Acknowledgments

For their valuable comments, discussions, and feedback, we wish to acknowledge Tim Jenkins, Kent Watsen, Susan Hares, Yang Geng, Peipei Guo, Michael Scharf, Sharon Chisholm, and Guangying Zheng.

9. References

9.1. Normative References

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

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

[RFC6470] Bierman, A., "Network Configuration Protocol (NETCONF) Base Notifications", RFC 6470, DOI 10.17487/RFC6470, February 2012, <https://www.rfc-editor.org/info/rfc6470>.

[RFC6536bis] Bierman, A. and M. Bjorklund, "Network Configuration Protocol (NETCONF) Access Control Model", draft-ietf-netconf-rfc6536bis-04 (work in progress), June 2017.

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

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

[RFC7951] Lhotka, L., "JSON Encoding of Data Modeled with YANG", RFC 7951, DOI 10.17487/RFC7951, August 2016, <https://www.rfc-editor.org/info/rfc7951>.

[RFC8072] Bierman, A., Bjorklund, M., and K. Watsen, "YANG Patch Media Type", RFC 8072, DOI 10.17487/RFC8072, February 2017, <https://www.rfc-editor.org/info/rfc8072>.

[subscribe]
Voit, E., Clemm, A., Gonzalez Prieto, A., Tripathy, A., and E. Nilsen-Nygaard, "Custom Subscription to Event Notifications", draft-ietf-netconf-subscribed-notifications-03 (work in progress), July 2017.

9.2. Informative References

[http-notif]
Voit, E., Gonzalez Prieto, A., Tripathy, A., Nilsen-Nygaard, E., Clemm, A., and A. Bierman, "Restconf and HTTP Transport for Event Notifications", August 2017.

[netconf-notif]
Gonzalez Prieto, A., Clemm, A., Voit, E., Nilsen-Nygaard, E., and A. Tripathy, "NETCONF Support for Event Notifications", July 2017.

[notifications2]
Voit, E., Bierman, A., Clemm, A., and T. Jenkins, "YANG Notification Headers and Bundles", July 2017.

[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, <https://www.rfc-editor.org/info/rfc6241>.

[RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, DOI 10.17487/RFC7223, May 2014, <https://www.rfc-editor.org/info/rfc7223>.

[RFC7923] Voit, E., Clemm, A., and A. Gonzalez Prieto, "Requirements for Subscription to YANG Datastores", RFC 7923, DOI 10.17487/RFC7923, June 2016, <https://www.rfc-editor.org/info/rfc7923>.

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

Appendix A. Relationships to other drafts

There are other related drafts which are progressing in the NETCONF WG. This section details the relationship of this draft to those others.

A.1. ietf-netconf-subscribed-notifications

The draft [subscribe] is the techical foundation around which the rest of the YANG push datastore specific mechanisms are layered.

A.2. ietf-netconf-netconf-event-notif

The [netconf-notif] draft supports yang-push by defining NETCONF transport specifics. Included are:

- bindings for RPC communications and Event Notifications over NETCONF.

- encoded examples

A.3. ietf-netconf-restconf-notif

The [http-notif] draft supports yang-push by defining transport specific guidance where some form of HTTP is used underneath. Included are:

- bindings for RPC communications over RESTCONF

- bindings for Event Notifications over HTTP2 and HTTP1.1

- encoded examples

- end-to-end deployment guidance for Call Home and TLS Heartbeat

A.4. voit-notifications2

The draft [notifications2] is not required to implement yang-push. Instead it defines data plane notification elements which improve the delivered experience. The following capabilities are specified:

- Defines common encapsulation headers objects to support functionality such as event severity, message signing, message loss discovery, message de-duplication, originating process identification.

- Defines how to bundle multiple event records into a single notification message.

These capabilities would be delivered by adding the drafts newly proposed header objects to the push-update and push-change-update notifications defined here. This draft is not yet adopted by the NETCONF WG.

Appendix B. Technologies to be considered for future iterations

B.1. Proxy YANG Subscription when the Subscriber and Receiver are different

The properties of Dynamic and Configured Subscriptions can be combined to enable deployment models where the Subscriber and Receiver are different. Such separation can be useful with some combination of:

- An operator does not want the subscription to be dependent on the maintenance of transport level keep-alives. (Transport independence provides different scalability characteristics.)

- There is not a transport session binding, and a transient Subscription needs to survive in an environment where there is unreliable connectivity with the Receiver and/or Subscriber.

- An operator wants the Publisher to include highly restrictive capacity management and Subscription security mechanisms outside of domain of existing operational or programmatic interfaces.

To build a Proxy Subscription, first the necessary information must be signaled as part of the <establish-subscription>. Using this set of Subscriber provided information; the same process described within section 3 will be followed.

After a successful establishment, if the Subscriber wishes to track the state of Receiver subscriptions, it may choose to place a separate on-change Subscription into the "Subscriptions" subtree of the YANG Datastore on the Publisher.

B.2. OpState and Filters

Currently the concept of datastores is undergoing a transformation. A new Network Management Datastore Architecture (NMDA) is currently being defined that will allow for improved handling and distinction of intended versus applied configurations. It will also extend the notion of a datastore to operational data. For implementations that are NMDA compliant in the future, some of the objects that are currently defined as operational data will no longer be required because they will in effect be redundant. Specifically, this concerns many of the objects in the read-only subscriptions branch of the tree. By omitting those objects, the model could be optimized further. However, there is no harm in the redundant objects and they allow for common management of YANG-Push subscriptions whether or not the underlying platform is fully NMDA-compliant, as opposed to

exposing client applications to heterogeneity in platform NMDA capabilities.

It is conceivable that filters are defined that apply to metadata, such as data nodes for which metadata has been defined that meets a certain criteria.

It is also conceivable to define in the future filters that are applied to actual data node values. Doing so would allow applications to subscribe to updates that are sent only when object values meet certain criteria, such as being outside a certain range.

Defining any such subscription filters at this point would be highly speculative in nature. However, it should be noted that corresponding extensions may be defined in future specifications. Any such extensions will be straightforward to accommodate by introducing a model that defines new filter types, and augmenting the new filter type into the subscription model.

B.3. Splitting push updates

Push updates may become fairly large and extend across multiple subsystems in a YANG-Push Server. As a result, it conceivable to not combine all updates into a single update message, but to split updates into multiple separate update messages. Such splitting could occur along multiple criteria: limiting the number of data nodes contained in a single update, grouping updates by subtree, grouping updates by internal subsystems (e.g., by line card), or grouping them by other criteria.

Splitting updates bears some resemblance to fragmenting packets. In effect, it can be seen as fragmenting update messages at an application level. However, from a transport perspective, splitting of update messages is not required as long as the transport does not impose a size limitation or provides its own fragmentation mechanism if needed. We assume this to be the case for YANG-Push. In the case of NETCONF, RESTCONF, HTTP/2, no limit on message size is imposed. In case of other transports, any message size limitations need to be handled by the corresponding transport mapping.

There may be some scenarios in which splitting updates might still make sense. For example, if updates are collected from multiple independent subsystems, those updates could be sent separately without need for combining. However, if updates were to be split, other issues arise. Examples include indicating the number of updates to the receiver, distinguishing a missed fragment from a missed update, and the ordering with which updates are received. Proper addressing those issues would result in considerable

complexity, while resulting in only very limited gains. In addition, if a subscription is found to result in updates that are too large, a publisher can always reject the request for a subscription while the subscriber is always free to break a subscription up into multiple subscriptions.

B.4. Potential Subscription Parameters

A possible is the introduction of an additional parameter "changes-only" for periodic subscription. Including this flag would results in sending at the end of each period an update containing only changes since the last update (i.e. a change-update as in the case of an on-change subscription), not a full snapshot of the subscribed information. Such an option might be interesting in case of data that is largely static and bandwidth-constrained environments.

Appendix C. Changes between revisions

(To be removed by RFC editor prior to publication)

v07 - v08

- Updated YANG models with minor tweaks to accommodate changes of ietf-subscribed-notifications.

v06 - v07

- Clarifying text tweaks.
- Clarification that filters act as selectors for subscribed data nodes; support for value filters not included but possible as a future extension
- Filters don't have to be matched to existing YANG objects

v05 - v06

- Security considerations updated.
- Base YANG model in [sn] updated as part of move to identities, YANG augmentations in this doc matched up
- Terms refined and text updates throughout
- Appendix talking about relationship to other drafts added.
- Datastore replaces stream

o Definitions of filters improved

v04 to v05

o Referenced based subscription document changed to Subscribed Notifications from 5277bis.

o Getting operational data from filters

o Extension notifiable-on-change added

o New appendix on potential futures. Moved text into there from several drafts.

o Subscription configuration section now just includes changed parameters from Subscribed Notifications

o Subscription monitoring moved into Subscribed Notifications

o New error and hint mechanisms included in text and in the yang model.

o Updated examples based on the error definitions

o Groupings updated for consistency

o Text updates throughout

v03 to v04

o Updates-not-sent flag added

o Not notifiable extension added

o Dampening period is for whole subscription, not single objects

o Moved start/stop into rfc5277bis

o Client and Server changed to subscriber, publisher, and receiver

o Anchor time for periodic

o Message format for synchronization (i.e. synch-on-start)

o Material moved into 5277bis

o QoS parameters supported, by not allowed to be modified by RPC

o Text updates throughout

Authors' Addresses

Alexander Clemm
Huawei

Email: ludwig@clemm.org

Eric Voit
Cisco Systems

Email: evoit@cisco.com

Alberto Gonzalez Prieto
VMware

Email: agonzalezpri@vmware.com

Ambika Prasad Tripathy
Cisco Systems

Email: ambtripa@cisco.com

Einar Nilsen-Nygaard
Cisco Systems

Email: einarnn@cisco.com

Andy Bierman
YumaWorks

Email: andy@yumaworks.com

Balazs Lengyel
Ericsson

Email: balazs.lengyel@ericsson.com