           Requirements for mounting of local and remote YANG subtrees
                draft-voit-netmod-yang-mount-requirements-00

Abstract

   Applications want simple ways to reference and access YANG objects
   and subtrees. These simplifications might include aliasing of local
   YANG information. These simplifications might include remote
   referencing of YANG information distributed across network.

   For such applications, development complexity is a barrier to YANG
   usage and therefore must be minimized. Specific aspects of
   complexity developers want to ignore include:

   o whether context specific aliases and paths to the same information
      can be exposed on a single device,

   o whether authoritative information is actually sourced from local
      or remote datastores,

   o whether the application needs to manage the overhead of session
      establishment and maintenance in order to access information on
      remote datastores,

   o whether objects have been locally cached or not, and

   o whether there is a mix of controllers, NMSs, and/or CLI which have
      access permission to update the primary copy of a particular
      object.

   The solution requirements described in this document detail what is
   needed to support application access to authoritative network YANG
   objects locally (via aliasing), or remotely from controllers or
   peering network devices in such a way to meet these goals.

Status of This Memo

Table of Contents

1. Business Problem

   Users, applications, and operators are asking for the ability to
   interact with local and remote information exposed as simply as
   possible from a familiar local datastore. Achieving an easy, local
   abstract representation of any information can be difficult since
   local YANG datastores might have been designed for alternative
   deployment contexts. Additionally for remote YANG objects, a running
   network is comprised of a distributed mesh of object ownership which
   can complicate effective YANG object addressing and retrieval.
   Solutions require the transparent assembly of local and remote
   objects in order to provide context specific, time synchronized, and
   consistent views required for a simple local abstraction.

   Ultimately local and network based application programming must be
   simplified to address these issues. To do this:

   o we must allow local and remote aliasing of network objects so that
      programmers can work against models which have been tuned for
      their development environment, structured in ways that best make
      sense to them

   o we must provide APIs to both controller and network element based
      applications in a way which allows access to these objects,

   o we must hide the mesh of interdependencies and consistency
      enforcement mechanisms between devices which will underpin a
      particular abstraction,

   o we must enable flexible deployment models, in which applications
      are able to run not only on controller and OSS frameworks but also
      on network devices without requiring heavy middleware with large
      footprints, and

o we need to maintain clear authoritative ownership of individual
   data items while not burdening applications with the need to
   reconcile and synchronize information replicated in different
   systems, nor needing to maintain redundant data models that
   operate on the same underlying data.

These steps will eliminate much unnecessary overhead currently
required of today's network programmer.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in RFC 2119 [RFC2119].

Alias Mount - A type of YANG Mount which provides an alternative path
to local objects of YANG data.

Authoritative Datastore - A datastore containing the authoritative
copy of an object, i.e. the source and the "owner" of the object.

Client Datastore - a datastore containing an object whose source and
"owner" is a remote datastore.

Data Node - An instance of management information in a YANG
datastore.

Datastore - A conceptual store of instantiated 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.

Mount Client - The system at which the mount point resides, into
which one or more subtrees may be mounted.

Mount Binding - An instance of YANG mount from a specific Mount Point
to a datastore. Types include:

o On-demand: Mount Client only pulls information when application
   requests

o Periodic: Mount Server pushes current state at a pre-defined
   interval

o Unsolicited: Mount Server maintains active bindings and sends to
   client cache upon change

Mount Point - Point in the local data store which may reference a
single remote subtree

Mount Server - The server with which the Mount Client communicates
and which provides the Mount Client with access to the mounted
information. Can be used synonymously with Mount Target.

Peer Mount - A type of YANG Mount which enables access to remote
objects as if they were contained within a local datastore
dynamically.

Schema Mount: A type of YANG Mount where a new YANG Schema is
constructed by inserting any existing YANG schema under a parent
model within a local datastore. Objects populated into the mounted
schema are only instantiated as part of the parent's hierarchy.

Target Data Node - Data Node on Mount Server against which a Mount
Binding is established

YANG Mount - The abstract concept of incorporating a YANG-defined
data tree or schema tree (the mounted data or schema tree) into a
existing YANG-defined data tree or schema tree (the parent data
tree).

3. Solution Context

YANG modeling has emerged as a preferred way to offer network
abstractions. The requirements in this document can be enabled by
expanding of the syntax of YANG capabilities embodied within RFC 6020
[RFC6020] and YANG 1.1 [rfc6020bis]. A companion draft to this one
which details a potential set of YANG technology extensions which can
support key requirements within this document are contained in .
[draft-clemm-mount].

To date systems built with full compliance to IETF YANG RFCs have
been missing two capabilities:

1. YANG Mount: Datastores have not been able to proxy objects
    located elsewhere on the same device, or upon a different device.
    This puts additional burden upon applications which then need to
    find and access multiple locations and which may be on remote
    systems.

2. Eventual Consistency: YANG Datastore implementations have
    typically assumed ACID [1] transaction models. There is nothing
    inherent in YANG itself which demands ACID transactional
    guarantees. YANG models can also expose information which might
    be in the process of undergoing convergence. Since IP networking

has been designed with convergence in mind, this is a useful
capability since some types of applications must participate
where there is dynamically changing state.

3.1. YANG Mount

First this document will dive deeper into YANG Datastore Mount
(a.k.a., "YANG Mount"). There are three subtypes of YANG Mount:
"Alias Mount", "Peer Mount", and "Schema Mount".

Alias Mount allows access to the same YANG data node along different
paths within the same YANG datastore, allowing a given subtree to
subtend from different YANG models within the same system. This
provides a means to:

o Provide application developers with custom and consolidated YANG
   objects that expose only the needed objects.

o Expose the objects organized into alternative structures,
   referenced via alternative application-intuitive paths. (This may
   include aliasing additional hierarchy layers to get to existing
   objects, including objects that had hitherto been right under
   root.)

o Accomplishing this without requiring mirroring or replication of
   the underlying data across various datastores.

Considering there are YANG models incorporating intersected and
replicated information today, adding an Alias Mount capability should
reduce YANG model development and model mapping requirements.

For Peer Mount, we need the capability to refer to managed resources
that reside on different systems. This allows applications on the
same system as the YANG datastore server, as well as remote clients
that access the datastore through a management protocol such as
NETCONF, to access all data from a familiar local YANG model.

o This is done in a manner that is transparent to users and
   applications.

o This is done in a way which does not require a user or application
   to be aware of the fact that some data resides in a different
   location and have them directly access that other system

In this way, an application developer is projected an image of one
virtual consolidated datastore. Peer Mount builds on Alias Mount by
allowing to incorporate redirection to remote systems into the
structure.

Schema Mount allows reuse of existing model definitions to facilitate implementation of alternative model structures in multiple contexts. In effect, it allows the definition of models which reuse other model definitions as if they had been defined as a special kind of grouping. As Schema Mount is being defined in drafts like [draft-bjorklund], and as Schema Mount details the Mounting of Schemas and not existing object data additional details will not be provided in this document.

So looking at the combination of Alias Mount and Peer Mount, the value to developers comes from its under-the-covers federation. A datastore using these capabilities transparently exposes information about objects that can be reached along multiple paths, allowing to make the same data nodes part of multiple concurrent hierarchies. The user does not need to be aware of the precise distribution and ownership of data themselves, nor is there a need for the application to discover those data sources, maintain separate associations with them, and partition its operations to fit along remote system boundaries. The effect is that a network device can broaden and customize the information available for local access. Life for the application is easier.

At the same time, the authoritative ownership of a data node is never in question. The original hierarchy and path that was defined when the data node was first defined in a YANG module remain in effect, and any validation involved in creating, modifying, or deleting the data node always occurs in the same context in which it was originally introduced. What such mounting allows is the definition of alternative, additional paths and hierarchies to which the object could also be accessed.

Any Object or subtree type can be exposed via such a Peer or Alias Mount reference. This can include configuration data that is either persistent or ephemeral, and which is valid within only a single device or across a domain of devices. This can include operational data that represents state across a single device or across a multiple devices.

A useful aspect of all types of YANG Mount is its ability to embed information from existing into newly defined models without requiring additional normalization effort. Normalization is a good thing, but the massive human efforts invested in uber-data-models have never gained industry traction due to the resulting models' brittle nature and complexity. By mounting subtrees/objects/schemas into local datastores it is possible to expose objects under a locally optimized hierarchy without having to worry about things such as transposing remote objects into a separate local model.

It should be noted that no variants of YANG Mount require knowledge
of the entire subtree being mounted. For example, there might be
augmentations of that subtree, or even mounted information in the
subtree itself. Likewise, mounted objects might dynamically change,
or even come into being. These dynamic changes can be reflected as
needed under the "attachment points" within the namespace hierarchy
where the data subtrees from remote systems have been mounted. In
this case, the precise details of what these subtrees exactly contain
does not need to be understood by the system implementing the
attachment point, it simply acts as a single point of entry and
"proxy" for the attached data.

3.2. Eventual Consistency and YANG

The CAP theorem [2] states that it is impossible for a distributed
computer system to simultaneously provide Consistency, Availability,
and Partition tolerance. (I.e., distributed network state management
is hard.) Mostly for this reason YANG implementations have shied
away from distributed datastore implementations where ACID
transactional guarantees cannot be given. This of course limits the
universe of applicability for YANG technology.

Leveraging YANG concepts, syntax, and models for objects which might
be happening to undergo network convergence is valuable. Such reuse
greatly expands the universe of information visible to networking
applications. The good news is that there is nothing in YANG syntax
that prohibits its reapplication for distributed datastores.
Extensions are needed however.

Requirements described within this document can be used to define
technology extensions to YANG 1.1 for remote datastore mounting.
Because of the CAP theorem, it must be recognized that systems built
upon these extensions MAY choose to support eventual consistency
rather than ACID guarantees. Some applications do not demand ACID
guarantees (examples are contained in this document's Use Case
section). Therefore for certain classes of applications, eventual
consistency [3] should be viewed as a cornerstone feature capability
rather than a bug.

Other industries have been able to identify and realize the value in
such model. The Object Management Group Data-Distribution Service
for Real-Time Systems has even standardized these capabilities for
non-YANG deployments [OMG-DDS]. Commercial deployments exist.

4. **Example Use Cases**

   Example Use Cases for Alias Mount can easily be seen from the
   description within Section 3.1. Therefore these are not detailed
   within this document. In general, those use cases involve imposing
   an alternative structure over YANG data models. YANG allows to
   extend and augment data models, allowing to add new data nodes as
   child nodes or as siblings to existing data nodes. However, YANG
   does not allow to superimpose a new data node on top of an existing
   one, or move an existing node under a newly defined node. Peer Mount
   closes that gap and allows to define models with alternative
   hierarchies and insert existing data nodes into that hierarchy.

   For Peer Mount, many types of applications can benefit from the
   simple and quick availability of objects from peer network devices.
   Because network management and orchestration systems have been
   fulfilling a subset of the requirements for decades, it is important
   to focus on what has changed. Changes include:

   o SDN applications wish to interact with local datastore(s) as if
      they represent the real-time state of the distributed network.

   o Independent sets of applications and SDN controllers might care
      about the same authoritative data node or subtree.

   o Changes in the real-time state of objects can announce themselves
      to subscribing applications.

   o The union of an ever increasing number of abstractions provided
      from different layers of the network are assumed to be consistent
      with each other (at least once a reasonable convergence time has
      been factored in).

   o CPU and VM improvements makes running Linux based applications on
      network elements viable.

   Such changes can enable a new class of applications. These
   applications are built upon fast-feedback-loops which dynamically
   tune the network based on iterative interactions upon a distributed
   datastore.

4.1. **Cloud Policer**

   A Cloud Policer enables a single aggregated data rate to tenants/
   users of a data center cloud that applies across their VMs; a rate
   independent of where specific VMs are physically hosted. This works
   by having edge router based traffic counters available to a
   centralized application, which can then maintain an aggregate across

those counters. Based on the sum of the counters across the set of
edge routers, new values for each device based Policer can be
recalculated and installed. Effectively policing rates are
continuously rebalanced based on the most recent traffic offered to
the aggregate set of edge devices.

The cloud policer provides a very simple cloud QoS model. Many other
QoS models could also be implemented. Example extensions include:

o CIR/PIR guarantees for a tenant,

o hierarchical QoS treatment,

o providing traffic delivery guarantees for specific enterprise
   branch offices, and

o adjusting the prioritization of one application based on the
   activity of another application which perhaps is in a completely
   different location.

It is possible to implement such a cloud policer application with
maximum application developer simplicity using peer mount. To do
this the application accesses a local datastore which in turn does a
peer mount from edge routers the objects which house current traffic
counter statistics. These counters are accessed as if they were part
of the local datastore structures, without concern for the fact that
the actual authoritative copies reside on remote systems.

Beyond this centralized counter collection peer mount, it is also
possible to have distributed edge routers mount information in the
reverse direction. In this case local edge routers can peer mount
centrally calculated policer rates for the device, and access these
objects as if they were locally configured.

For both directions of mounting, the authoritative copy resides in a
single system and is mounted by peers. Therefore issues with regards
to inconsistent configuration of the same redundant data across the
network are avoided. Also as can be seen in this use case, the same
system can act as a mount client of some objects while acting as
server for other objects.

4.2. DDoS Thresholding

Another extension of the "Cloud Policer" application is the creation
of additional action thresholds at bandwidth rates far greater than
might be expected. If these higher thresholds are hit, it is
possible to connect in DDoS scrubbers to ingress traffic. This can
be done in seconds after a bandwidth spike. This can also be done if

non-bandwidth counters are available. For example, if TCP flag
counts are available it is possible to look for changes in SYN/ACK
ratios which might signal a different type of attack. In all cases,
when network counters indicate a return to normal traffic profiles
the DDoS Scrubbers can be automatically disconnected.

Benefits of only connecting a DDoS scrubber in the rare event an
attack might be underway include:

o marking down traffic for an out-of-profile tenant so that an
   potential attack doesn't adversely impact others,

o applying DDoS Scrubbing across many devices when an attack is
   detected in one,

o reducing DDoS scrubber CPU, power, and licensing requirements
   (during the vast majority of time, spikes are not occurring), and

o dynamic management and allocation of scarce platform resources
   (such as optimizing span port usage, or limiting IP-FIX reporting
   to levels where devices can do full flow detail exporting).

4.3. Service Chain Classification, Load Balancing and Capacity Management

Service Chains will dynamically change ingress classification
filters, allocate paths from many ingress devices across shared
resources. This information needs to be updated in real time as
available capacity is allocated or failures are discovered. It is
possible to simplify service chain configuration and dynamic topology
maintenance by transparently updating remote cached topologies when
an authoritative object is changed within a central repository. For
example if the CPU in one VM spikes, you might want to recalculate
and adjust many chained paths to relieve the pressure. Or perhaps
after the recalculation you want to spin up a new VM, and then adjust
chains when that capacity is on-line.

A key value here is central calculation and transparent auto-
distribution. In other words, a change only need be updated by an
application in a single location, and the infrastructure will
automatically synchronize changes across any number of subscribing
devices without application involvement. In fact, the application
need not even know many devices are monitoring the object which has
been changed.

Beyond 1:n policy distribution, applications can step back from
aspects of failure recovery. What happens if a device is rebooting
or simply misses a distribution of new information? With peer mount

there is no doubt as to where the authoritative information resides
if things get out of synch.

While this ability is certainly useful for dynamic service chain
filtering classification and next hop mapping, this use case has more
general applicability. With a distributed datastore, diverse
applications and hosts can locally access a single device's current
VM CPU and Bandwidth values. They can do it without needing to
explicitly query that remote machine. Updates from a device would
come from a periodic push of stats to a transparent cache to
subscribed, or via an unsolicited update which is only sent when
these value exceed established norms.

5. Requirements

To achieve the objectives described above for Alias Mount and Peer
Mount, the network needs to support a number of requirements:

5.1. Application Simplification

A major obstacle to network programmability are any requirements
which force applications to use abstractions more complicated than
the developer cares to touch. To simplify applications development
and reduce unnecessary code, the following needs must be met.

Applications MUST be able to access a local datastore which includes
objects whose authoritative source perhaps is located in a elsewhere
in some datastore.

Local datastores MUST be able to provide a hierarchical view of
objects assembled from objects whose authoritative source may
potentially originate from different and overlapping namespaces.

Applications MUST be able to access all objects of a datastore
without concern where the actual object is located, i.e. whether the
authoritative copy of the object is hosted on the same system as the
local datastore or whether it is hosted in a remote datastore.

A datastore's application facing interfaces MUST make no
differentiation whether individual objects exposed are
authoritatively owned by the datastore or mounted from elsewhere

When a change is made to an object, that change will be reflected in
any datastore in which the object is included.

A datastore supporting Alias or Peer Mount MUST allow the same object
to be mounted from multiple places.

Applications SHOULD be able to extract a time synchronized set of
operational data from the datastore. (In other words, the
application asks for a subset of network state at time-stamp or time-
range "X". The datastore would then deliver time synchronized
snapshots of the network state per the request. The datastore may
work with NTP and operational counter to optimize the synchronization
results of such a query. It is understood that some types of data
might be undergoing convergence conditions.)

Authoritative datastore retain full ownership of "their" objects.
This means that while remote datastores may access the data, any
modifications to objects that are initiated at those remote
datastores need to be authorized by the authoritative owner of the
data. Likewise, the authoritative owner of the data may make changes
to objects, including modifications, additions, and deletions,
without needing to first ask for permission from remote clients.

Applications MUST be designed to deal with incomplete data if remote
objects are not accessible, e.g. due to temporal connectivity issues
preventing access to the authoritative source. (This will be true
for many protocols and programming languages. Mount is unlikely to
add anything new here unless applications have extra error handling
routines to deal with when there is no response from a remote
system.).

5.2. Caching

Remote objects in a datastore can be accessed "on demand", when the
application interacting with the datastore demands it. In that case,
a request made to the local datastore is forwarded to the remote
system. The response from the remote system, e.g. the retrieved
data, is subsequently merged and collated with the other data to
return a consolidated response to the invoking application.

A downside of a datastore which is distributed across devices can be
the latency induced when remote object acquisition is necessary.
There are plenty of applications which have requirements which simply
cannot be served when latency is introduced. The good news is that
the concept of caching lends itself well to distributed datastores.
It is possible to transparently store some types of objects locally
even when the authoritative copy is remote. Instead of fetching data
on demand when an application demands it, the application is simply
provided with the local copy. It is then up to the datastore
infrastructure to keep selected replicated info in synch, e.g. by
prefetching information, or by having the remote system publish
updates which are then locally stored. At this point, it is expected
that a preferred method of subscribing to and publishing updates will

be accomplished via [i2rs-pub-sub-reqts] and [yang-push]. Other
methods could work equally well .

This is not a new idea. Caching and Content Delivery Networks (CDN)
have sped read access for objects within the Internet for years.
This has enabled greater performance and scale for certain content.
Just as important, these technologies have been employed without end
user applications being explicitly aware of their involvement. Such
concepts are applicable for scaling the performance of a distributed
datastore.

Where caching occurs, it MUST be possible for the Mount Client to
store object copies of a remote data node or subtree in such a way
that applications are unaware that any caching is occurring.
However, the interface to a datastore MAY provide applications with a
special mode/flag to allow them to force a read-through.

Where caching occurs, system administration facilities SHOULD allow
facilities to flush either the entire cache, or information
associated with select Mount Points.

5.3. Subscribing to Remote Object Updates

When caching occurs, data can go stale. [yang-push] provides a
mechanism where changes in an authoritative data node or subtree can
be monitored. If changes occur, these changes can be delivered to
any subscribing datastores. In this way remote caches can be kept
up-to-date. In this way, directly monitoring remote applications can
quickly receive notifications without continuous polling.

A Mount Server SHOULD support [yang-push] Periodic and/or On-Change
pub/sub capabilities in which one or more remote clients subscribe to
updates of a target data node / subtree, which are then automatically
published by the Mount Server.

It MUST be possible for Applications to bind to subscribed Data Node
/ Subtrees so that upon Mount Client receipt of subscribed
information, it is immediately passed to the application.

It MUST be possible for a Target Data Node to support 1:n Mount
Bindings to many subscribed Mount Points.

5.4. Lifecycle of the Mount Topology

Mount can drive a dynamic and richly interconnected mesh of peer-to-
peer of object relationships. Each of these Mounts will be
independently established by a Mount Client.

It MUST be possible to bootstrap the Mount Client by providing the
YANG paths to resources on the Mount Server.

There SHOULD be the ability to add Mount Client bindings during run-
time.

A Mount Client MUST be able to be able to create, delete, and timeout
Mount Bindings.

Any Subscription MUST be able to inform the Mount Client of an
intentional/graceful disconnect.

A Mount Client MUST be able to verify the status of Subscriptions,
and drive re-establishment if it has disappeared.

5.4.1. Discovery and Creation of Mount Topology

Application visibility into an ever-changing set of network objects
is not trivial. While some applications can be easily configured to
know the Devices and available Mount Points of interest, other
applications will have to balance many aspects of dynamic device
availability, capabilities, and interconnectedness. Maintenance of
these dynamic elements can be done on the YANG objects themselves
without anything needed new for any type of YANG Mount.

5.4.2. Restrictions on the Mount Topology

Mount Clients MUST NOT create recursive Mount bindings (i.e., the
Mount Client should not load any object or subtree which it has
already delivered to another in the role of a Mount Server.) Note:
Objects mounted from a controller as part of orchestration are *not*
considered the same objects as those which might be mounted back from
a network device showing the actual running config.

5.5. Mount Filter

The Mount Server default MUST be to deliver the same Data Node /
Subtree that would have been delivered via direct YANG access.

It SHOULD be possible for a Mount Client to request something less
than the full subtree or a target node as defined in
[i2rs-pub-sub-reqts].

5.6. Auto-Negotiation of Peer Mount Client QoS

The interest that a Mount Client expresses in a particular subtree
SHOULD include the non-functional data delivery requirements (QoS) on
the data that is being mounted. Additionally, Mount Servers SHOULD

advertise their data delivery capabilities. With this information
the Mount Client can decide whether the quality of the delivered data
is sufficient to serve applications residing above the Mount Client.

An example here is reliability. A reliable protocol might be
overkill for a state that is republished with high frequency.
Therefore a Mount Server may sometimes choose to not provide a
reliable method of communication for certain objects. It is up to
the Mount Client to determine whether what is offered is sufficiently
reliable for its application. Only when the Mount Server is offering
data delivery QoS better or equal to what is requested, shall a mount
binding be established.

Another example is where subscribed objects must be pushed from the
Mount Server within a certain interval from when an object change is
identified. In such a scenario the interval period of the Mount
Server must be equal or smaller than what is requested by a Mount
Client. If this "deadline" is not met by the Mount Server the
infrastructure MAY take action to notify clients.

5.7. Datastore Qualification

It is conceivable to differentiate between different datastores on
the remote server, that is, to designate the name of the actual
datastore to mount, e.g. "running" or "startup". If on the target
node there are multiple datastores available, but there has no
specific datastore identified by the Mount Client, then the running
or "effective" datastore is the assumed target.

It is conceivable to use such Datastore Qualification in conjunction
with ephemeral datastores, to address requirements being worked in
the I2RS WG [draft-i2rs-ephemeral].

5.8. Mount Cascades

It is possible for the mounted subtree to in turn contain a
mountpoint. However, circular mount relationships MUST NOT be
introduced. For this reason, a mounted subtree MUST NOT contain a
mountpoint that refers back to a mount target that directly or
indirectly contains the originating mountpoint. As part of a mount
operation, the mount points of the mounted system need to be checked
accordingly.

5.9. Transport

Many secured transports are viable assuming transport, data security,
scale, and performance objectives are met. Netconf and/or Restconf

should be considered as starting points. Other transports may be
proposed over time.

It MUST be possible to support Netconf or Restconf Transport of
subscribed Nodes and Subtrees.

5.10. Security Considerations

Many security mechanisms exist to protect data access for CLI and API
on network devices. To the degree possible these mechanisms should
transparently protect data when performing a Peer Mount.

The same mechanisms used to determine whether a remote host has
access to a particular YANG Data Node or Subtree MUST be invoked to
determine whether a Mount Client has access to that information.

The same traditional transport level security mechanism security used
for YANG over a particular transport MUST be used for the delivery of
objects from a Mount Server to a Mount Client.

A Mount Server implementation MUST NOT change any credentials passed
by the Mount Client system for any Mount Binding request.

The Mount Server MUST deliver no more objects from a Data Node or
Subtree than allowable based on the security credentials provided by
the Mount Client.

To ensure the ensuring maximum scale limits, it MUST be possible to
for a Mount Server to limit the number of bindings and transactional
limits

It SHOULD be possible to prioritize which Mount Binding instances
should be serviced first if there is CPU, bandwidth, or other
capacity constraints.

5.11. High Availability

A key intent for Peer Mount is to allow access to an authoritative
copy of an object for a particular domain. Of course system and
software failures or scheduled upgrades might mean that the primary
copy is not consistently accessible from a single device. In
addition, system failovers might mean that the authoritative copy
might be housed on a different device than the one where the binding
was originally established. Peer Mount architectures must be built
to enable Mount Clients to transparently provide access to objects
where the authoritative copy moves due to dynamic network
reconfigurations .

A Peer Mount architecture MUST guarantee that mount bindings between
a Mount Server and Mount Clients drive system behavior which is at
least eventually consistent. The infrastructure providing this level
of consistency MUST be able to operate in scenarios where a system is
(temporarily) not fully connected. Furthermore, Mount Clients MAY
have various requirements on the boundaries under which eventual
consistency is allowed to take place. This subject can be decomposed
in the following items:

5.11.1. Reliability

A scenario that deserves attention in particular is when a subset of
Mount Clients receive and cache a pushed subscription update. If a
Mount Server loses connectivity, cross network element consistency
can be lost. In such a scenario Mount Clients MAY elect a new
designated Mount Server from the set of Mount Clients which have
received the latest state.

5.11.2. Alignment to late joining peers

When a mount binding is established a Mount Server SHOULD provide the
Mount Client with the latest state of the requested data. In order
to increase availability and fault tolerance an infrastructure MAY
support the capability to have multiple alignment sources. In
(temporary) absence of a Mount Server, Mount Clients MAY elect a
temporary Mount Server to service late joining Mount Clients.

5.11.3. Liveliness

Upon losing liveliness and being unable to refresh cached data
provided from a Mount Server, a Mount Client MAY decide to purge the
mount bindings of that server. Purging mount bindings under such
conditions however makes a system vulnerable to losing network-wide
consistency. A Mount Client can take proactive action based on the
assumption that the Mount Server is no longer available. When
connectivity is only temporarily lost, this assumption could be false
for other datastores. This can introduce a potential for decision-
making based on semantical disagreement. To properly handle these
scenarios, application behavior MUST be designed accordingly and
timeouts with regards to liveliness detection MUST be carefully
determined.

5.11.4. Merging of datasets

A traditional problem with merging replicated datasets during the
failover and recovery of Mount Servers is handling the corresponding
target data node lifecycle management. When two replicas of a
dataset experienced a prolonged loss of connectivity a merge between

the two is required upon re-establishing connectivity. A replica
might have been modifying contents of the set, including deletion of
objects. A naive merge of the two replicas would discard these
deletes by aligning the now stale, deleted objects to the replica
that deleted them.

Authoritative ownership is an elegant solution to this problem since
modifications of content can only take place at the owner. Therefore
a Mount Client SHOULD, upon reestablishing connectivity with a newly
authoritative Mount Server, replace any existing cache contents from
a mount binding with the latest version.

5.11.5. Distributed Mount Servers

For selected objects, Mount Bindings SHOULD be allowed to Anycast
addresses so that a Distributed Mount Server implementation can
transparently provide (a) availability during failure events to Mount
Clients, and (b) load balancing on behalf of Mount Clients.

5.12. Configuration

At the Mount Client, it MUST be possible for all Mount bindings to
configure the following such that the application needs no knowledge.
This will include a diverse list of elements such as the YANG URI
path to the remote subtree.

5.13. Assurance and Monitoring

API usage for YANG should be tracked via existing mechanisms. There
is no intent to require additional transaction tracking than would
have been provided normally. However there are additional
requirements which should allow the state of existing and historical
bindings to be provided.

A Mount Client MUST be able to poll a Mount Server for the state of
Subsciptions maintained between the two devices.

A Mount Server MUST be able to publish the set of Subscriptions which
are currently established on or below any identified data node.

6. IANA Considerations

This document makes no request of IANA.

7. Acknowledgements

   We wish to acknowledge the helpful contributions, comments, and
   suggestions that were received from Ambika Prasad Tripathy. Shashi
   Kumar Bansal, Prabhakara Yellai, Dinkar Kunjikrishnan, Harish
   Gumaste, Rohit M., Shruthi V. , Sudarshan Ganapathi, and Swaroop
   Shastri.

8. References

8.1. Normative References

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

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

8.2. Informative References

   [draft-clemm-mount]
              Clemm, Alexander., "Mounting YANG-Defined Information from
              Remote Datastores", April 2015, <http://tools.ietf.org/id/
              draft-clemm-netmod-mount-03.txt>.

   [draft-i2rs-ephemeral]
              Haas, J., "I2RS Ephemeral State Requirements", March 2016,
              <http://tools.ietf.org/html/
              draft-ietf-i2rs-ephemeral-state-00>.

   [i2rs-pub-sub-reqts]
              Voit, Eric., Clemm, Alexander., and Alberto. Gonzalez
              Prieto, "Requirements for Subscription to YANG
              Datastores", February 2016,
              <http://datatracker.ietf.org/doc/
              draft-ietf-i2rs-pub-sub-requirements/>.

   [OMG-DDS] "Data Distribution Service for Real-time Systems, version
              1.2", January 2007, <http://www.omg.org/spec/DDS/1.2/>.

[rfc6020bis]

          Bjorklund, Martin., "YANG - A Data Modeling Language for
          the Network Configuration Protocol (NETCONF)", February
          2016, <https://tools.ietf.org/html/draft-ietf-netmod-
          rfc6020bis-05>.

[yang-push]

          Clemm, A., "Subscribing to datastore push updates",
          February 2016, <https://tools.ietf.org/html/draft-ietf-
          netconf-yang-push-01>.

8.3. URIs

   [1] http://en.wikipedia.org/wiki/ACID

   [2] http://robertgreiner.com/2014/08/cap-theorem-revisited/

   [3] http://guide.couchdb.org/draft/consistency.html

Authors' Addresses

   Eric Voit
   Cisco Systems

   Email: evoit@cisco.com

   Alexander Clemm
   Cisco Systems

   Email: alex@cisco.com

   Sander Mertens
   Prismtech

   Email: sander.mertens8@gmail.com