NETCONF Data Modeling Language Working Group (netmod)
Internet-Draft
Intended status: Informational
Expires: March 29, 2015

E. Voit
A. Clemm
S. Bansal
A. Tripathy
P. Yellai
Cisco Systems

September 25, 2014

Requirements for Peer Mounting of YANG subtrees from Remote Datastores
draft-voit-netmod-peer-mount-requirements-00

Abstract

Network integrated applications want simple ways to access YANG objects and subtrees which might be distributed across network. Performance requirements may dictate that it is unaffordable for a subset of these applications to go through existing centralized management brokers. For such applications, development complexity must be minimized. Specific aspects of complexity developers want to ignore include:

o whether authoritative information is actually sourced from remote datastores (as well as how to get to those datastores),

o whether such information has been locally cached or not,

o whether there are zero, one, or more controllers asserting ownership of information, and

o whether there are interactions with other applications concurrently running elsewhere

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

Status of This Memo

Copyright Notice

Table of Contents

1. Business Problem

Instrumenting Physical and Virtual Network Elements purely along device boundaries is insufficient for today’s requirements. Instead, users, applications, and operators are asking for the ability to interact with varying subsets of network information at the highest viable level of abstraction. Likewise applications that run locally on devices may require access to data that transcends the boundaries of the device they are deployed. Achieving this can be difficult since a running network is comprised of a distributed mesh of object ownership. (I.e., the authoritative device owning a particular object will vary.) Solutions require the transparent assembly of different objects from across a network in order to provide consolidated, time synchronized, and consistent views required for that abstraction.

Recent approaches have focused on a Network Controller as the arbiter of new network-wide abstractions. Controller based solutions are supportable by requirements outlined in this document. However this is not the only deployment model covered by this document. Equally valid are deployment models where Network Elements exchange information in a way which allows one or more of those Elements to provide the desired network level abstraction. This is not a new idea. Examples of Network Element based protocols which already do network level abstractions include VRRP [RFC3768], mLACP/ICCP[ICCP], and Anycast-RP [RFC4610] . As network elements increase their compute power and support Linux based compute virtualization, we should expect additional local applications to emerge as well (such as Distributed Analytics [1]).

Ultimately network application programming must be simplified. To do this:

- we must provide APIs to both controller and network element based applications in a way which allows access to network objects as if they were coming from a cloud,

- we must enable these local applications to interact with network level abstractions,

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

- 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

- 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].

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 on or more remote subtrees may be mounted.

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

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

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

- 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 - The act of representing remote objects in the local datastore

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

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].A "-02" release of this draft which includes specifications to support many additional concepts will be posted in the coming days.

To date systems built upon YANG models have been missing two capabilities:

1. Peer Datastore Mount: Datastores have not been able to proxy objects located elsewhere. This puts additional burden upon applications which then need to find and access multiple (potentially remote) systems.

2. Eventual Consistency: YANG Datastore implementations have typically assumed ACID [2] 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. Peer Mount

First this document will dive deeper into Peer Datastore Mount (a.k.a., "Peer Mount"). Contrary to existing YANG datastores, where hierarchical datatree(s) are local in scope and only includes data that is "owned" by the local system, we need an agent or interface on one system which is able refer to managed resources that reside on another system. 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 as if it were local to that same server. This must be done in a manner that is transparent to users and applications. This must be 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, the user is projected an image of one virtual consolidated datastore.

The value in such a datastore comes from its under-the-covers federation. The datastore transparently exposes information from multiple systems across the network. 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.

Any Object type can be included in such a datastore. 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.

Another useful aspect of "Peer Mount" is its ability to embed information from external YANG models which haven’t necessarily been normalized. 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 remote trees/objects into local datastores it is possible to expose remote objects under a locally optimized hierarchy without having to transpose remote objects into a separate local model. Once this exists, object translation and normalization become optional capabilities which may also be hidden.

Another useful aspect of "Peer Mount" is its ability to mount remote trees where the local datastore does not know the full subtree being installed. In fact, the remote datastore might be dynamically changing the mounted tree. 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 1.1

The CAP theorem [3] 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 1.1 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 [4] should be viewed as a cornerstone feature capability rather than a bug.

4. Example Use Cases

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:

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

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

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

- 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).

- 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:

- CIR/PIR guarantees for a tenant,

- hierarchical QoS treatment,

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

- 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:

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

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

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

- 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, 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 is located in a remote datastore hosted on a different server.

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

With two exceptions, a datastore’s application facing interfaces MUST make no differentiation whether individual objects exposed are authoritatively owned by the datastore or mounted from remote. This includes Netconf and Restconf as well as other, possibly proprietary interfaces (such as, CLI generated from corresponding YANG data models). The two exceptions are that it is acceptable to make a distinction between an object authoritatively owned by the data store and a remote object as follows:

o Object updates / editing, creation and deletion. E.g. via edit-config conditions and constraints are assessed at the authoritative datastore when the update/create/delete is conducted. Any conditions or constraints at remote client datastores are NOT assessed.

o Locks obtained at a client datastore: It is conceivable for the interface to distinguish between two lock modes: locking the entire subtree including remote data (in which case the datastore’s mount client needs to explicitly obtain and release locks from mounted authoritative datastores), or locking only authoritatively owned data, excluding remote data from the lock.

These exceptions should not be very problematic as non-authoritative copies will typically be marked as read-only. This will not violate any considerations of "no differentiation" of local or remote.

When a change is made to an object, that change will be reflected in any datastore in which the object is included. This means that a change made to the object through a remote datastore will affect the object in the authoritative datastore. Likewise, changes to an object in the authoritative datastore will be reflected at any client datastores.

The distributed datastore MUST be able to include objects from multiple remote datastores. The same object may be included in multiple remote datastores; in other words, an object’s authoritative datastore MUST support multiple clients.

The distributed datastore infrastructure MUST enable to access to some subset of the same objects on different devices. (This includes multiple controllers as well as multiple physical and virtual peer devices.)

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 Considerations

5.2.1. Caching Overview

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.

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 and perhaps even a write-through.

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

5.2.2. Pub/Sub of Object Updates

When caching occurs, data can go stale. Pub/Sub 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.

5.2.2.1. General Pub/Sub Update Requirements

A Mount Client SHOULD be able to take advantage of pub/sub capabilities offered by a mount server. However, not every Mount Server offers according capabilities.

A Mount Client SHOULD be able to revert back to retrieve objects "On Demand" and/or to pre-fetch objects by request.

A Mount Server MAY support a pub/sub capability 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.

One or more of the following pub/sub policies MUST be supported:

- On Demand (i.e. no pub/sub) - default

- Periodic (with a specified time interval)

- On change, immediate as the change occurs.

- On change, at the end of fixed intervals if a change has occurred

Further modifications are possible: e.g. on change, whether to only publish only the object that has changed or the entire subtree that had been subscribed to. (Effectively this is aggregate replication at tree level, not at the object level.)

Pub/sub is applicable to other applications as well, not limited to peer mounting. For example, a pub/sub capability can greatly facilitate monitoring, as applications no longer have to "poll" for data but can simply choose to subscribe to a stream of the most current data. Accordingly, servers that offer pub/sub capabilities for its YANG datastore SHOULD NOT limit subscribers to Mount Clients, but allow other applications to subscribe as well.

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

It MUST be possible for the Mount Client to subscribe to Data Node / Subtrees so that upon Mount Client receipt of subscribed information, it is cached and therefore awaiting local application requests.

If there are no applications subscribing to a Data Node / Subtree, a server SHOULD cease to publish the corresponding data.

It MUST be possible for a Subscription to include a timestamp when the Subscription will expire.

It MUST be possible to identify a specific time when a Mount Binding will return the current value(s) of a mounted Data Node / Subtree. (Such timeframes can be in the very near future in order to support a snapshot of network state or counters across many devices.)

A publisher is not responsible to monitor if the subscribers are still active. It MAY do so, but is not obliged to do so. Subscriptions upon a Target Data Node do not remain active forever but MUST be periodically re-subscribed . The reason for this is to avoid "waste", for example in cases when subscribers "die". If a subscriber restarts, it is the subscribers responsibility to check whether its subscriptions are still intact or to resubscribe if needed.

It MUST be possible for a Target Data Node to support 1:1 Mount Bindings to a single subscribed Mount Point.

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

5.2.2.2. Periodic Pub/Sub Updates

Especially with network based Counters or Operational data, there need be no recurring request to send the next instance of data which is released on schedule to subscribers.

It MUST be possible to for a Periodic Mount Point to identify a specific time when a Mount Target will return the current value(s) of a mounted Data Node / Subtree. This will allow for synchronization of calculation for objects delivered from many Mount Bindings to local applications.

It MUST be possible to for a Periodic Mount Point to identify the desired start and stop timestamps for any replicated objects associated with duration. This will allow for time period synchronization of source data for objects delivered from many Mount Bindings to local applications.

5.2.2.3. Change-trigger Pub/Sub Updates

For an Unsolicited Mount Point, if a data node or subtree changes, the Mount Target MUST provide updated objects to the Mount Client.

For an Unsolicited Mount Point, if a data node or subtree changes, the Mount Target SHOULD be able to provide just the updated objects to the Mount Client. Note: If there is a Mount Filter in place, then only the updated objects based on the filter will be delivered. It is possible that a Filter will result in no update needing to be sent.

It SHOULD be possible to provide criteria per Mount Binding on the characteristics of changes to a Target Data Node’s monitored objects on before an update is sent to the subscribing system. (Effectively this becomes a "threshold trigger" for change notification to remote caches.)

5.3. 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.

A Mount Server maintaining a periodic or unsolicited Mount Binding MUST be able to inform the Mount Client of an intentional graceful disconnection of that binding.

A Mount Client must be able to verify the existence of a periodic or unsolicited Mount Binding which has successfully been established on a Mount Server, and re-establish if it has disappeared.

5.3.1. 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. For the most part, maintenance of these dynamic elements can be done on the YANG objects themselves without anything needed new for Peer Mount. Technologies such as need reference are covered in other standards initiatives. Therefore this draft does delve deeply into the needs for Auto-discovery of YANG objects which may be advertised.

However it will likely become interesting for a network element to limit the Data Subtrees which might be subscribed for Unsolicited and Periodic Update.

It SHOULD be possible for a Mount Server to advertise potential Target Data Nodes which can support unsolicited and periodic binding types.

5.3.2. 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.

5.4. 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 that the full subtree or a target node. This will be valuable when the number or size of objects under a Target Data Node is large.

5.5. Transport

Many secured transports are viable assuming transport, data security, scale, and performance objectives are met. Netconf is recommended for starting. Other transports may be proposed over time. Additional study is needed to assess how aspects of locking will supported in parallel with eventual consistency for different object writes.

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

RESTconf [RESTconf] must be examined as well, especially as section 1.2 studies a possible mix of locking.

5.6. Security Considerations

Many security mechanisms exist to protect read/write access for CLI and API on network devices. To the degree possible these mechanisms should transparently protect data read and write when performing a Peer Mount. The text below starts with a subset of those requirements . Additional ones should be added.

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.

5.7. 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 .

For selected objects, Mount Bindings SHOULD be allowed to Anycast or ECMP (Equal Cost Multiple Path) 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.

Where anycast unsolicited or periodic bindings are allowed to Anycast addresses, the real time state of Mount Server bindings MUST be coordinated across the set of Anycast addressed devices. In this way, the state of periodic and unsolicited Mount Bindings will not be lost during a failover.

The Mount Client and Mount Server MUST either have heart-beat mechanism OR use a connection oriented transport to detect each other’s failures.

When a Mount Server detects disappearance of a Mount Client, the Mount Server SHOULD purge all the mount bindings from the failed Mount Client.

When a failover occurs on the Mount Client side, the new instance of the Mount Client SHOULD re-establish the mount bindings with the Mount Server(s).

When a failover occurs on the Mount Server side, the new owner of an unsolicited mount binding SHOULD send out the current state of the object to subscribed Mount Clients.

5.8. 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 includea diverse list of elements such at the YANG URI path to the remote subtree.

5.9. 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 unsolicited and periodic Mount Binding maintained between the two devices.

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

A Mount Server MUST be able to publish the set of unsolicited and periodic Mount Bindings which are going to a specific Mount Client.

A Mount Server MUST be able to publish the set fulfilled Mount Bindings which are going to a specific Mount Client.

A Mount Server MUST be able to publish a list of the Mount Bindings transactions successfully completed.

A Mount Server MUST be able to publish a list of the Mount Bindings which failed, along with reasons that they failed. These reasons might include:

- Improper security credentials provided for the Mount Client to access the target node
- Target node referenced does not exist
- Binding type requested not available for the target node
- Mount Server out of resources or resources not available
- Connection from client lost before binding complete

A Mount Client MUST be able to publish a list of the Mount Bindings transactions successfully completed.

A Mount Client MUST be able to publish a list of the Mount Bindings which failed, along with reasons that they failed. These reasons might include:

- No response from Mount Client
- Connection could not be established with Mount Client
- Security credentials provided to Mount Server rejected
- Target node referenced does not exist
- Binding type requested not available for the target node
- Mount Server out of resources or resources not available
- Connection from client lost before binding complete

6. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

7. Acknowledgements

We wish to acknowledge the helpful contributions, comments, and suggestions that were received from 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, March 1997.

[RFC3768] Hinden, R., "Virtual Router Redundancy Protocol (VRRP)", RFC 3768, April 2004.

[RFC4610] Farinacci, D. and Y. Cai, "Anycast-RP Using Protocol Independent Multicast (PIM)", RFC 4610, August 2006.

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

8.2. Informative References

[ICCP] Martini, L., Ed., "Inter-Chassis Communication Protocol for L2VPN PE Redundancy", March 2014, <https://tools.ietf.org/html/draft-ietf-pwe3-iccp-16>.

[RESTconf] Bierman, A., Ed., "RESTCONF Protocol", July 2014, <https://tools.ietf.org/html/draft-ietf-netconf-restconf-01>.

[draft-clemm-mount] Clemm, A., Ed., "Mounting YANG-Defined Information from Remote Datastores", September 2013, <http://tools.ietf.org/id/draft-clemm-netmod-mount-01.txt>.

[rfc6020bis] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", July 2014, <https://datatracker.ietf.org/doc/draft-ietf-netmod-rfc6020bis/>.

8.3. URIs

[1] http://thomaswdinsmore.com/2014/05/01/distributed-analytics-primer/

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

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

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

Authors’ Addresses

Eric Voit
Cisco Systems

Email: evoit@cisco.com

Alex Clemm
Cisco Systems

Email: alex@cisco.com

Shashi Kumar Bansal
Cisco Systems

Email: shabansa@cisco.com

Ambika Tripathy
Cisco Systems

Email: ambtripa@cisco.com

Prabhakara Yellai
Cisco Systems

Email: pyellai@cisco.com