Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 22, 2016

A. Clemm
J. Medved
E. Voit
Cisco Systems

March 21, 2016

# Mounting YANG-Defined Information from Remote Datastores
## draft-clemm-netmod-mount-04.txt

Abstract

This document introduces capabilities that allow YANG datastores to reference and incorporate information from remote datastores. This is accomplished by extending YANG with the ability to define mount points that reference data nodes in another YANG subtree, by subsequently allowing those data nodes to be accessed by client applications as if part of an alternative data hierarchy, and by providing the necessary means to manage and administer those mount points. Two flavors are defined: Alias-Mount allows to mount local subtrees, while Peer-Mount allows subtrees to reside on and be authoritatively owned by a remote server. YANG-Mount facilitates the development of applications that need to access data that transcends individual network devices while improving network-wide object consistency, or that require an aliasing capability to be able to create overlay structures for YANG data.

Table of Contents

1. Introduction . . . . . 3
1.1. Overview . . . . . 3
1.2. Examples . . . . . 5
2. Definitions and Acronyms . . . . . 7
3. Example scenarios . . . . . 8
3.1. Network controller view . . . . . 8
3.2. Consistent network configuration . . . . . 10
4. Operating on mounted data . . . . . 11
4.1. General principles . . . . . 12
4.2. Data retrieval . . . . . 12
4.3. Other operations . . . . . 13
4.4. Other considerations . . . . . 13
5. Data model structure . . . . . 14
5.1. YANG mountpoint extensions . . . . . 14
5.2. YANG structure diagrams . . . . . 15
5.3. Mountpoint management . . . . . 15
5.4. Caching . . . . . 17
5.5. Other considerations . . . . . 18
5.5.1. Authorization . . . . . 18

5.5.2. Datastore qualification . . . . . 18
5.5.3. Mount cascades . . . . . 19
5.5.4. Implementation considerations . . . . . 19
5.5.5. Modeling best practices . . . . . 20
6. Datastore mountpoint YANG module . . . . . 20
7. Security Considerations . . . . . 28
8. Acknowledgements . . . . . 29
9. References . . . . . 29
9.1. Normative References . . . . . 29
9.2. Informative References . . . . . 30
Appendix A. Example . . . . . 31
Authors' Addresses . . . . . 35

# 1. Introduction

## 1.1. Overview

This document introduces a new capability that allows YANG datastores [RFC6020] to incorporate and reference information from other YANG subtrees. The capability allows a client application to retrieve and have visibility of that YANG data as part of an alternative structure. This is provided by introducing a mountpoint concept. This concept allows to declare a YANG data node in a primary datastore to serve as a "mount point" under which a subtree with YANG data can be mounted. This way, data nodes from another subtree can be inserted into an alternative data hierarchy, arranged below local data nodes. To the user, this provides visibility to data from other subtrees, rendered in a way that makes it appear largely as if it were an integral part of the datastore. This enables users to retrieve local "native" as well as mounted data in integrated fashion, using e.g. Netconf [RFC6241] or Restconf [I-D.ietf-netconf-restconf] data retrieval primitives. The concept is reminiscent of concepts in a Network File System that allows to mount remote folders and make them appear as if they were contained in the local file system of the user's machine.

Two variants of YANG-Mount are introduced, which build on one another:

- Alias-Mount allows mountpoints to reference a local YANG subtree residing on the same server. It provides effectively an aliasing capability, allowing for an alternative hierarchy and path for the same YANG data.

- Peer-Mount allows mountpoints to reference a remote YANG subtree, residing on a different server. It can be thought of as an extension to Alias-Mount, in which a remote server can be

specified. Peer-Mount allows a server to effectively provide a federated datastore, including YANG data from across the network.

In each case, mounted data is authoritatively owned by the server that it is a part of. Validation of integrity constraints apply to the authoritative copy; mounting merely provides a different view of the same data. It does not impose additional constraints on that same data; however, mounted data may be referred to from other data nodes. The mountpoint concept applies in principle to operations beyond data retrieval, i.e. to configuration, RPCs, and notifications. However, support for such operations involves additional considerations, for example if support for configuration transactions and locking (which might now apply across the network) were to be provided. While it is conceivable that additional capabilities for operations on mounted information are introduced at some point in time, their specification is beyond the scope of this specification.

YANG does provide means by which modules that have been separately defined can reference and augment one another. YANG also does provide means to specify data nodes that reference other data nodes. However, all the data is assumed to be instantiated as part of the same datastore, for example a datastore provided through a NETCONF server. Existing YANG mechanisms do not account for the possibility that some information that needs to be referred not only resides in a different subtree of the same datastore, or was defined in a separate module that is also instantiated in the same datastore, but that is genuinely part of a different datastore that is provided by a different server.

The ability to mount information from local and remote datastores is new and not covered by existing YANG mechanisms. Until now, management information provided in a datastore has been intrinsically tied to the same server and to a single data hierarchy. In contrast, the capability introduced in this specification allows the server to render alternative data hierarchies, and to represent information from remote systems as if it were its own and contained in its own local data hierarchy.

The capability of allowing the mounting of information from other subtrees is accomplished by a set of YANG extensions that allow to define such mount points. For this purpose, a new YANG module is introduced. The module defines the YANG extensions, as well as a data model that can be used to manage the mountpoints and mounting process itself. Only the mounting module and its server (i.e. the "receivers" or "consumers" of the mounted information) need to be aware of the concepts introduced here. Mounting is transparent to the "providers" of the mounted information and models that are being

mounted; any data nodes or subtrees within any YANG model can be mounted.

Alias-Mount and Peer-Mount build on top of each other. It is possible for a server to support Alias-Mount but not Peer-Mount. In essence, Peer-Mount requires an additional parameter that is used to refer to the target system. This parameter does not need to be supported if only Alias-Mount is provided.

Finally, it should be mentioned that Alias-Mount and Peer-Mount are not to be confused with Schema-Mount [I-D.bjorklund-netmod-structural-mount]. Schema Mount allows to instantiate an existing model definition underneath a mount point, not reference a set of YANG data that has already been instantiated somewhere else. In that sense, Schema-Mount resembles more a "grouping" concept that allows to reuse an existing definition in a new context, as opposed to referencing and incorporating existing instance information into a new context.

## 1.2. Examples

The requirements for mounting YANG subtrees from remote datastores, as long as a set of associated use cases, are documented in [I-D.voit-netmod-yang-mount-requirements]. The ability to mount data from remote datastores is useful to address various problems that several categories of applications are faced with.

One category of applications that can leverage this capability are network controller applications that need to present a consolidated view of management information in datastores across a network. Controller applications are faced with the problem that in order to expose information, that information needs to be part of their own datastore. Today, this requires support of a corresponding YANG data module. In order to expose information that concerns other network elements, that information has to be replicated into the controller's own datastore in the form of data nodes that may mirror but are clearly distinct from corresponding data nodes in the network element's datastore. In addition, in many cases, a controller needs to impose its own hierarchy on the data that is different from the one that was defined as part of the original module. An example for this concerns interface data, both operational data (e.g. various types of interface statistics) and configuration data, such as defined in [RFC7223]. This data will be contained in a top-level container ("interfaces", in this particular case) in a network element datastore. The controller may need to provide its clients a view on interface data from multiple devices under its scope of control. One way of to do so would involve organizing the data in a list with separate list elements for each device. However, this in

turn would require introduction of redundant YANG modules that effectively replicate the same interface data save for differences in hierarchy.

By directly mounting information from network element datastores, the controller does not need to replicate the same information from multiple datastores, nor does it need to re-define any network element and system-level abstractions to be able to put them in the context of network abstractions. Instead, the subtree of the remote system is attached to the local mount point. Operations that need to access data below the mount point are in effect transparently redirected to remote system, which is the authoritative owner of the data. The mounting system does not even necessarily need to be aware of the specific data in the remote subtree. Optionally, caching strategies can be employed in which the mounting system prefetches data.

A second category of applications concerns decentralized networking applications that require globally consistent configuration of parameters. When each network element maintains its own datastore with the same configurable settings, a single global change requires modifying the same information in many network elements across a network. In case of inconsistent configurations, network failures can result that are difficult to troubleshoot. In many cases, what is more desirable is the ability to configure such settings in a single place, then make them available to every network element. Today, this requires in general the introduction of specialized servers and configuration options outside the scope of NETCONF, such as RADIUS [RFC2866] or DHCP [RFC2131]. In order to address this within the scope of NETCONF and YANG, the same information would have to be redundantly modeled and maintained, representing operational data (mirroring some remote server) on some network elements and configuration data on a designated master. Either way, additional complexity ensues.

Instead of replicating the same global parameters across different datastores, the solution presented in this document allows a single copy to be maintained in a subtree of single datastore that is then mounted by every network element that requires awareness of these parameters. The global parameters can be hosted in a controller or a designated network element. This considerably simplifies the management of such parameters that need to be known across elements in a network and require global consistency.

It should be noted that for these and many other applications merely having a view of the remote information is sufficient. It allows to define consolidated views of information without the need for replicating data and models that have already been defined, to audit

information, and to validate consistency of configurations across a network. Only retrieval operations are required; no operations that involve configuring remote data are involved.

## 2. Definitions and Acronyms

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

DHCP: Dynamic Host Configuration Protocol.

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

Datastore-push: A mechanism that allows a client to subscribe to updates from a datastore, which are then automatically pushed by the server to the client.

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 the remote subtree is mounted.

Mount point: A data node that receives the root node of the remote datastore being mounted.

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.

Mount target: A remote server whose datastore is being mounted.

NACM: NETCONF Access Control Model

NETCONF: Network Configuration Protocol

RADIUS: Remote Authentication Dial In User Service.

RPC: Remote Procedure Call

Remote datastore: A datastore residing at a remote node.

URI: Uniform Resource Identifier

YANG: A data definition language for NETCONF

## 3. Example scenarios

The following example scenarios outline some of the ways in which the ability to mount YANG datastores can be applied. Other mount topologies can be conceived in addition to the ones presented here.

### 3.1. Network controller view

Network controllers can use the mounting capability to present a consolidated view of management information across the network. This allows network controllers to expose network-wide abstractions, such as topologies or paths, multi-device abstractions, such as VRRP [RFC3768], and network-element specific abstractions, such as information about a network element's interfaces.

While an application on top of a controller could bypass the controller to access network elements directly for their element-specific abstractions, this would come at the expense of added inconvenience for the client application. In addition, it would compromise the ability to provide layered architectures in which access to the network by controller applications is truly channeled through the controller.

Without a mounting capability, a network controller would need to at least conceptually replicate data from network elements to provide such a view, incorporating network element information into its own controller model that is separate from the network element's, indicating that the information in the controller model is to be populated from network elements. This can introduce issues such as data inconsistency and staleness. Equally important, it would lead to the need to define redundant data models: one model that is implemented by the network element itself, and another model to be implemented by the network controller. This leads to poor maintainability, as analogous information has to be redundantly defined and implemented across different data models. In general, controllers cannot simply support the same modules as their network elements for the same information because that information needs to be put into a different context. This leads to "node"-information that needs to be instantiated and indexed differently, because there are multiple instances across different data stores.

For example, "system"-level information of a network element would most naturally placed into a top-level container at that network element's datastore. At the same time, the same information in the context of the overall network, such as maintained by a controller, might better be provided in a list. For example, the controller might maintain a list with a list element for each network element, underneath which the network element's system-level information is

contained. However, the containment structure of data nodes in a module, once defined, cannot be changed. This means that in the context of a network controller, a second module that repeats the same system-level information would need to be defined, implemented, and maintained. Any augmentations that add additional system-level information to the original module will likewise need to be redundantly defined, once for the "system" module, a second time for the "controller" module.

By allowing a network controller to directly mount information from network element datastores, the controller does not need to replicate the same information from multiple datastores. Perhaps even more importantly, the need to re-define any network element and system-level abstractions just to be able to put them in the context of network abstractions is avoided. In this solution, a network controller's datastore mounts information from many network element datastores. For example, the network controller datastore (the "primary" datastore) could implement a list in which each list element contains a mountpoint. Each mountpoint mounts a subtree from a different network element's datastore. The data from the mounted subtrees is then accessible to clients of the primary datastore using the usual data retrieval operations.

This scenario is depicted in Figure 1. In the figure, M1 is the mountpoint for the datastore in Network Element 1 and M2 is the mountpoint for the datastore in Network Element 2. MDN1 is the mounted data node in Network Element 1, and MDN2 is the mounted data node in Network Element 2.

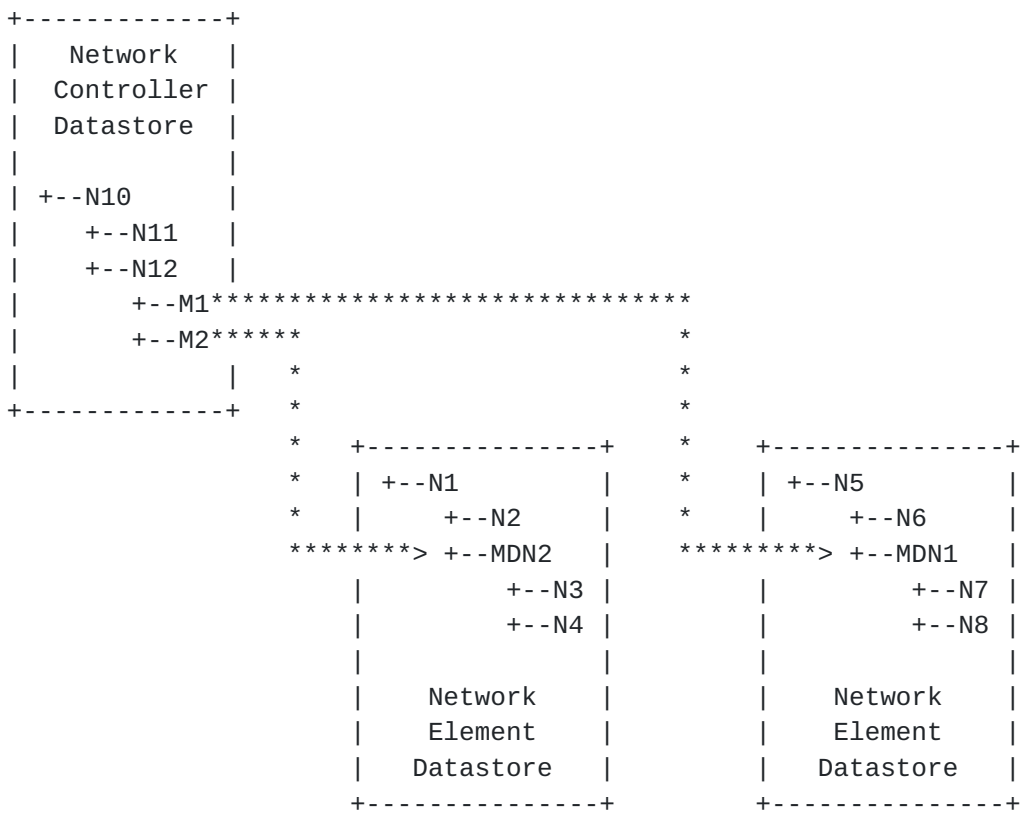


Figure 1: Network controller mount topology

## 3.2. Consistent network configuration

A second category of applications concerns decentralized networking applications that require globally consistent configuration of parameters that need to be known across elements in a network. Today, the configuration of such parameters is generally performed on a per network element basis, which is not only redundant but, more importantly, error-prone. Inconsistent configurations lead to erroneous network behavior that can be challenging to troubleshoot.

Using the ability to mount information from remote datastores opens up a new possibility for managing such settings. Instead of replicating the same global parameters across different datastores, a single copy is maintained in a subtree of single datastore. This datastore can hosted in a controller or a designated network element. The subtree is subsequently mounted by every network element that requires access to these parameters.

In many ways, this category of applications is an inverse of the previous category: Whereas in the network controller case data from many different datastores would be mounted into the same datastore with multiple mountpoints, in this case many elements, each with

their own datastore, mount the same remote datastore, which is then mounted by many different systems.

The scenario is depicted in Figure 2. In the figure, M1 is the mountpoint for the Network Controller datastore in Network Element 1 and M2 is the mountpoint for the Network Controller datastore in Network Element 2. MDN is the mounted data node in the Network Controller datastore that contains the data nodes that represent the shared configuration settings. (Note that there is no reason why the Network Controller Datastore in this figure could not simply reside on a network element itself; the division of responsibilities is a logical one.

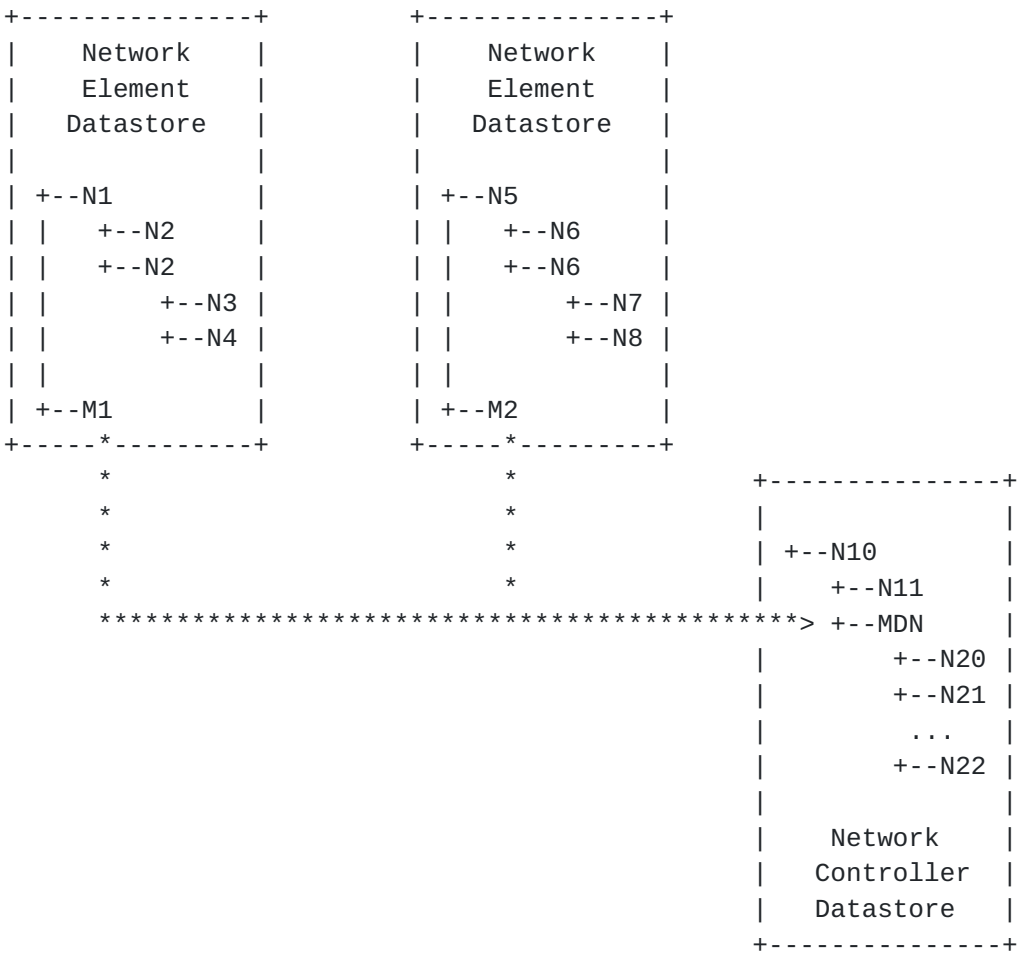


Figure 2: Distributed config settings topology

## 4. Operating on mounted data

This section provides a rough illustration of the operations flow involving mounted datastores.

### 4.1. General principles

The first thing that should be noted about these operations flows concerns the fact that a mount client essentially constitutes a special management application that interacts with a subtree to render the data of that subtree as an alternative tree hierarchy. In the case of Alias-Mount, both original and alternative tree are maintained by the same server, which in effect provides alternative paths to the same data. In the case of Peer-Mount, the mount client constitutes in effect another application, with the remote system remaining the authoritative owner of the data. While it is conceivable that the remote system (or an application that proxies for the remote system) provides certain functionality to facilitate the specific needs of the mount client to make it more efficient, the fact that another system decides to expose a certain "view" of that data is fundamentally not the remote system's concern.

When a client application makes a request to a server that involves data that is mounted from a remote system, the server will effectively act as a proxy to the remote system on the client application's behalf. It will extract from the client application request the portion that involves the mounted subtree from the remote system. It will strip that portion of the local context, i.e. remove any local data paths and insert the data path of the mounted remote subtree, as appropriate. The server will then forward the transposed request to the remote system that is the authoritative owner of the mounted data, acting itself as a client to the remote server. Upon receiving the reply, the server will transpose the results into the local context as needed, for example map the data paths into the local data tree structure, and combine those results with the results of the remainder portion of the original request.

### 4.2. Data retrieval

Data retrieval operations are the only category of operations that is supported for peer-mounted information. In that case, a Netconf "get" or "get-configuration" operation might be applied on a subtree whose scope includes a mount point. When resolving the mount point, the server issues its own "get" or "get-configuration" request against the remote system's subtree that is attached to the mount point. The returned information is then inserted into the data structure that is in turn returned to the client that originally invoked the request.

### 4.3. Other operations

The fact that only data retrieval operations are the only category of operations that are supported for peer-mounted information does not preclude other operations to be applied to datastore subtrees that contain mountpoints and peer-mounted information. Peer-mounted information is simply transparent to those operations. When an operation is applied to a subtree which includes mountpoints, mounted information is ignored for purposes of the operation. For example, for a Netconf "edit-config" operation that includes a subtree with a mountpoint, a server will ignore the data under the mountpoint and apply the operation only to the local configuration. Mounted data is "read-only" data. The server does not even need to return an error message that the operation could not be applied to mounted data; the mountpoint is simply ignored.

In principle, it is conceivable that operations other than data-retrieval are applied to mounted data as well. For example, an operation to edit configuration information might expect edits to be applied to remote systems as part of the operation, where the edited subtree involves mounted information. However, editing of information and "writing through" to remote systems potentially involves significant complexity, particularly if transactions and locking across multiple configuration items are involved. Support for such operations will require additional capabilities, specification of which is beyond the scope of this specification.

Likewise, YANG-Mount does not extend towards RPCs that are defined as part of YANG modules whose contents is being mounted. Support for RPCs that involve mounted portions of the datastore, while conceivable, would require introduction of an additional capability, whose definition is outside the scope of this specification.

By the same token, YANG-Mount does not extend towards notifications. It is conceivable to offer such support in the future using a separate capability, definition of which is once again outside the scope of this specification.

### 4.4. Other considerations

Since mounting of information typically involves communication with a remote system, there is a possibility that the remote system will not respond within a certain amount of time, that connectivity is lost, or that other errors occur. Accordingly, the ability to mount datastores also involves mountpoint management, which includes the ability to configure timeouts, retries, and management of mountpoint state (including dynamic addition removal of mountpoints). Mountpoint management will be discussed in section Section 5.3.

It is expected that some implementations will introduce caching schemes. Caching can increase performance and efficiency in certain scenarios (for example, in the case of data that is frequently read but that rarely changes), but increases implementation complexity. Caching is not required for YANG-mount to work - in which case access to mounted information is "on-demand", in which the authoritative data node always gets accessed. Whether to perform caching is a local implementation decision.

When caching is introduced, it can benefit from the ability to subscribe to updates on remote data by remote servers. Requirements for such a capability have been defined in [I-D.ietf-i2rs-pub-sub-requirements]. Some optimizations to facilitate caching support will be discussed in section Section 5.4.

## 5. Data model structure

### 5.1. YANG mountpoint extensions

At the center of the module is a set of YANG extensions that allow to define a mountpoint.

- The first extension, "mountpoint", is used to declare a mountpoint. The extension takes the name of the mountpoint as an argument.

- The second extension, "subtree", serves as substatement underneath a mountpoint statement. It takes an argument that defines the root node of the datastore subtree that is to be mounted, specified as string that contains a path expression. This extension is used to define mountpoints for Alias-Mount, as well as Peer-Mount.

- The third extension, "target", also serves as a substatement underneath a mountpoint statement. It is used for Peer-Mount and takes an argument that identifies the target system. The argument is a reference to a data node that contains the information that is needed to identify and address a remote server, such as an IP address, a host name, or a URI [RFC3986].

A mountpoint MUST be contained underneath a container. Future revisions might allow for mountpoints to be contained underneath other data nodes, such as lists, leaf-lists, and cases. However, to keep things simple, at this point mounting is only allowed directly underneath a container.

Only a single data node can be mounted at one time. While the mount target could refer to any data node, it is recommended that as a best

practice, the mount target SHOULD refer to a container. It is possible to maintain e.g. a list of mount points, with each mount point each of which has a mount target an element of a remote list. However, to avoid unnecessary proliferation of the number of mount points and associated management overhead, when data from lists or leaf-lists is to be mounted, a container containing the list respectively leaf-list SHOULD be mounted instead of individual list elements.

It is possible for a mounted datastore to contain another mountpoint, thus leading to several levels of mount indirections. However, mountpoints MUST NOT introduce circular dependencies. In particular, a mounted datastore MUST NOT contain a mountpoint which specifies the mounting datastore as a target and a subtree which contains as root node a data node that in turn contains the original mountpoint. Whenever a mount operation is performed, this condition mountpoint. Whenever a mount operation is performed, this condition MUST be validated by the mount client.

## 5.2. YANG structure diagrams

YANG data model structure overviews have proven very useful to convey the "Big Picture". It would be useful to indicate in YANG data model structure overviews the fact that a given data node serves as a mountpoint. We propose for this purpose also a corresponding extension to the structure representation convention. Specifically, we propose to prefix the name of the mounting data node with upper-case 'M'.

```
rw network
+-- rw nodes
    +-- rw node [node-ID]
        +-- rw node-ID
        +-- M node-system-info
```

## 5.3. Mountpoint management

The YANG module contains facilities to manage the mountpoints themselves.

For this purpose, a list of the mountpoints is introduced. Each list element represents a single mountpoint. It includes an identification of the mount target, i.e. the remote system hosting the remote datastore and a definition of the subtree of the remote data node being mounted. It also includes monitoring information about current status (indicating whether the mount has been successful and is operational, or whether an error condition applies

such as the target being unreachable or referring to an invalid subtree).

In addition to the list of mountpoints, a set of global mount policy settings allows to set parameters such as mount retries and timeouts.

Each mountpoint list element also contains a set of the same configuration knobs, allowing administrators to override global mount policies and configure mount policies on a per-mountpoint basis if needed.

There are two ways how mounting occurs: automatic (dynamically performed as part of system operation) or manually (administered by a user or client application). A separate mountpoint-origin object is used to distinguish between manually configured and automatically populated mountpoints.

Whether mounting occurs automatically or needs to be manually configured by a user or an application can depend on the mountpoint being defined, i.e. the semantics of the model.

When configured automatically, mountpoint information is automatically populated by the datastore that implements the mountpoint. The precise mechanisms for discovering mount targets and bootstrapping mount points are provided by the mount client infrastructure and outside the scope of this specification. Likewise, when a mountpoint should be deleted and when it should merely have its mount-status indicate that the target is unreachable is a system-specific implementation decision.

Manual mounting consists of two steps. In a first step, a mountpoint is manually configured by a user or client application through administrative action. Once a mountpoint has been configured, actual mounting occurs through an RPCs that is defined specifically for that purpose. To unmount, a separate RPC is invoked; mountpoint configuration information needs to be explicitly deleted. Manual mounting can also be used to override automatic mounting, for example to allow an administrator to set up or remove a mountpoint.

It should be noted that mountpoint management does not allow users to manually "extend" the model, i.e. simply add a subtree underneath some arbitrary data node into a datastore, without a supporting mountpoint defined in the model to support it. A mountpoint definition is a formal part of the model with well-defined semantics. Accordingly, mountpoint management does not allow users to dynamically "extend" the data model itself. It allows users to populate the datastore and mount structure within the confines of a model that has been defined prior.

The structure of the mountpoint management data model is depicted in the following figure, where brackets enclose list keys, "rw" means configuration, "ro" operational state data, and "?" designates optional nodes. Parantheses enclose choice and case nodes. The figure does not depict all definitions; it is intended to illustrate the overall structure.

```
module: ietf-mount
   +--rw mount-server-mgmt {mount-server-mgmt}?
      +--rw mountpoints
      |  +--rw mountpoint* [mountpoint-id]
      |     +--rw mountpoint-id        string
      |     +--ro mountpoint-origin?   enumeration
      |     +--rw subtree-ref          subtree-ref
      |     +--rw mount-target
      |     |  +--rw (target-address-type)
      |     |     +--:(IP)
      |     |     |  +--rw target-ip?          inet:ip-address
      |     |     +--:(URI)
      |     |     |  +--rw uri?                inet:uri
      |     |     +--:(host-name)
      |     |     |  +--rw hostname?           inet:host
      |     |     +--:(node-ID)
      |     |     |  +--rw node-info-ref?      subtree-ref
      |     |     +--:(other)
      |     |        +--rw opaque-target-ID?   string
      |     +--ro mount-status?        mount-status
      |     +--rw manual-mount?        empty
      |     +--rw retry-timer?         uint16
      |     +--rw number-of-retries?   uint8
      +--rw global-mount-policies
         +--rw manual-mount?        empty
         +--rw retry-timer?         uint16
         +--rw number-of-retries?   uint8
```

## 5.4. Caching

Under certain circumstances, it can be useful to maintain a cache of remote information. Instead of accessing the remote system, requests are served from a copy that is locally maintained. This is particularly advantageous in cases where data is slow changing, i.e. when there are many more "read" operations than changes to the underlying data node, and in cases when a significant delay were incurred when accessing the remote system, which might be prohibitive for certain applications. Examples of such applications are applications that involve real-time control loops requiring response times that are measured in milliseconds. However, as data nodes that are mounted from an authoritative datastore represent the "golden

copy", it is important that any modifications are reflected as soon as they are made.

It is a local implementation decision of mount clients whether to cache information once it has been fetched. However, in order to support more powerful caching schemes, it becomes necessary for the mount server to "push" information proactively. For this purpose, it is useful for the mount client to subscribe for updates to the mounted information at the mount server. A corresponding mechanism that can be leveraged for this purpose is specified in [I-D.ietf-netconf-yang-push].

Note that caching large mountpoints can be expensive. Therefore limiting the amount of data unnecessarily passed when mounting near the top of a YANG subtree is important. For these reasons, an ability to specify a particular caching strategy in conjunction with mountpoints can be desirable, including the ability to exclude certain nodes and subtrees from caching. According capabilities may be introduced in a future version of this draft.

## 5.5. Other considerations

### 5.5.1. Authorization

Access to mounted information is subject to authorization rules. To the mounted system, a mounting client will in general appear like any other client. Authorization privileges for remote mounting clients need to be specified through NACM (NETCONF Access Control Model) [RFC6536].

### 5.5.2. 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". However, for the purposes of this spec, we assume that the datastore to be mounted is generally implied. Mounted information is treated as analogous to operational data; in general, this means the running or "effective" datastore is the target. That said, the information which targets to mount does constitute configuration and can hence be part of a startup or candidate datastore.

It is conceivable to use mount in conjunction with ephemeral datastores, to address requirements outlined in [I-D.haas-i2rs-netmod-netconf-requirements]. Support for such a scheme is for further study and may be included in a future revision of this spec.

### 5.5.3. 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 the mounting system with 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.

### 5.5.4. Implementation considerations

Implementation specifics are outside the scope of this specification. That said, the following considerations apply:

Systems that wish to mount information from remote datastores need to implement a mount client. The mount client communicates with a remote system to access the remote datastore. To do so, there are several options:

- The mount client acts as a NETCONF client to a remote system. Alternatively, another interface to the remote system can be used, such as a REST API using JSON encodings, as specified in [I-D.ietf-netconf-restconf]. Either way, to the remote system, the mount client constitutes essentially a client application like any other. The mount client in effect IS a special kind of client application.

- The mount client communicates with a remote mount server through a separate protocol. The mount server is deployed on the same system as the remote NETCONF datastore and interacts with it through a set of local APIs.

- The mount client communicates with a remote mount server that acts as a NETCONF client proxy to a remote system, on the client's behalf. The communication between mount client and remote mount server might involve a separate protocol, which is translated into NETCONF operations by the remote mount server.

It is the responsibility of the mount client to manage the association with the target system, e.g. validate it is still reachable by maintaining a permanent association, perform reachability checks in case of a connectionless transport, etc.

It is the responsibility of the mount client to manage the mountpoints. This means that the mount client needs to populate the mountpoint monitoring information (e.g. keep mount-status up to data and determine in the case of automatic mounting when to add and

remove mountpoint configuration). In the case of automatic mounting, the mount client also interacts with the mountpoint discovery and bootstrap process.

The mount client needs to also participate in servicing datastore operations involving mounted information. An operation requested involving a mountpoint is relayed by the mounting system's infrastructure to the mount client. For example, a request to retrieve information from a datastore leads to an invocation of an internal mount client API when a mount point is reached. The mount client then relays a corresponding operation to the remote datastore. It subsequently relays the result along with any responses back to the invoking infrastructure, which then merges the result (e.g. a retrieved subtree with the rest of the information that was retrieved) as needed. Relaying the result may involve the need to transpose error response codes in certain corner cases, e.g. when mounted information could not be reached due to loss of connectivity with the remote server, or when a configuration request failed due to validation error.

### 5.5.5. Modeling best practices

There is a certain amount of overhead associated with each mount point. The mount point needs to be managed and state maintained. Data subscriptions need to be maintained. Requests including mounted subtrees need to be decomposed and responses from multiple systems combined.

For those reasons, as a general best practice, models that make use of mount points SHOULD be defined in a way that minimizes the number of mountpoints required. Finely granular mounts, in which multiple mountpoints are maintained with the same remote system, each containing only very small data subtrees, SHOULD be avoided. For example, lists SHOULD only contain mountpoints when individual list elements are associated with different remote systems. To mount data from lists in remote datastores, a container node that contains all list elements SHOULD be mounted instead of mounting each list element individually. Likewise, instead of having mount points refer to nodes contained underneath choices, a mountpoint should refer to a container of the choice.

## 6. Datastore mountpoint YANG module

```
<CODE BEGINS>
file "ietf-mount@2016-03-21.yang"
module ietf-mount {
  namespace "urn:ietf:params:xml:ns:yang:ietf-mount";
  prefix mnt;
```

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

     organization
       "IETF NETMOD (NETCONF Data Modeling Language) Working Group";
     contact
       "WG Web:   <http://tools.ietf.org/wg/netmod/>
        WG List:  <mailto:netmod@ietf.org>

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

        WG Chair: Lou Berger
                  <mailto:lberger@labn.net>

        WG Chair: Juergen Schoenwaelder
                  <mailto:j.schoenwaelder@jacobs-university.de>

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

        Editor: Jan Medved
        <mailto:jmedved@cisco.com>

        Editor: Eric Voit
        <mailto:evoit@cisco.com>";
     description
       "This module provides a set of YANG extensions and definitions
        that can be used to mount information from remote datastores.";

     revision 2016-03-21 {
       description
         "Initial revision.";
       reference
         "draft-clemm-netmod-mount-04.txt";
     }

     extension mountpoint {
       argument name;
       description
         "This YANG extension is used to mount data from another
          subtree in place of the node under which this YANG extension
          statement is used.

          This extension takes one argument which specifies the name
          of the mountpoint.
```

```
        This extension can occur as a substatement underneath a
        container statement, a list statement, or a case statement.
        As a best practice, it SHOULD occur as statement only
        underneath a container statement, but it MAY also occur
        underneath a list or a case statement.

        The extension can take two parameters, target and subtree,
        each defined as their own YANG extensions.

        For Alias-Mount, a mountpoint statement MUST contain a
        subtree statement for the mountpoint definition to be valid.
        For Peer-Mount, a mountpoint statement MUST contain both a
        target and a subtree substatement for the mountpoint
        definition to be valid.

        The subtree SHOULD be specified in terms of a data node of
        type 'mnt:subtree-ref'. The targeted data node MUST
        represent a container.

        The target system MAY be specified in terms of a data node
        that uses the grouping 'mnt:mount-target'.  However, it
        can be specified also in terms of any other data node that
        contains sufficient information to address the mount target,
        such as an IP address, a host name, or a URI.

        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 the mounting system
        with a mount target that directly or indirectly contains the
        originating mountpoint.";
    }

    extension target {
      argument target-name;
      description
        "This YANG extension is used to perform a Peer-Mount.
        It is used to specify a remote target system from which to
        mount a datastore subtree.  This YANG
        extension takes one argument which specifies the remote
        system. In general, this argument will contain the name of
        a data node that contains the remote system information. It
        is recommended that the reference data node uses the
        mount-target grouping that is defined further below in this
        module.

        This YANG extension can occur only as a substatement below
        a mountpoint statement. It MUST NOT occur as a substatement
```

```
          below any other YANG statement.";
      }

      extension subtree {
        argument subtree-path;
        description
          "This YANG extension is used to specify a subtree in a
           datastore that is to be mounted.  This YANG extension takes
           one argument which specifies the path to the root of the
           subtree. The root of the subtree SHOULD represent an
           instance of a YANG container.  However, it MAY represent
           also another data node.

           This YANG extension can occur only as a substatement below
           a mountpoint statement. It MUST NOT occur as a substatement
           below any other YANG statement.";
      }

      feature mount-server-mgmt {
        description
          "Provide additional capabilities to manage remote mount
           points";
      }

      typedef mount-status {
        type enumeration {
          enum "ok" {
            description
              "Mounted";
          }
          enum "no-target" {
            description
              "The argument of the mountpoint does not define a
               target system";
          }
          enum "no-subtree" {
            description
              "The argument of the mountpoint does not define a
                root of a subtree";
          }
          enum "target-unreachable" {
            description
              "The specified target system is currently
               unreachable";
          }
          enum "mount-failure" {
            description
              "Any other mount failure";
```

```
        }
        enum "unmounted" {
          description
            "The specified mountpoint has been unmounted as the
             result of a management operation";
        }
      }
      description
        "This type is used to represent the status of a
         mountpoint.";
    }

    typedef subtree-ref {
      type string;
      description
        "This string specifies a path to a datanode. It corresponds
         to the path substatement of a leafref type statement.  Its
         syntax needs to conform to the corresponding subset of the
         XPath abbreviated syntax. Contrary to a leafref type,
         subtree-ref allows to refer to a node in a remote datastore.
         Also, a subtree-ref refers only to a single node, not a list
         of nodes.";
    }

    grouping mount-monitor {
      description
        "This grouping contains data nodes that indicate the
         current status of a mount point.";
      leaf mount-status {
        type mount-status;
        config false;
        description
          "Indicates whether a mountpoint has been successfully
           mounted or whether some kind of fault condition is
           present.";
      }
    }

    grouping mount-target {
      description
        "This grouping contains data nodes that can be used to
         identify a remote system from which to mount a datastore
         subtree.";
      container mount-target {
        description
          "A container is used to keep mount target information
           together.";
        choice target-address-type {
```

```
          mandatory true;
          description
            "Allows to identify mount target in different ways,
             i.e. using different types of addresses.";
          case IP {
            leaf target-ip {
              type inet:ip-address;
              description
                "IP address identifying the mount target.";
            }
          }
          case URI {
            leaf uri {
              type inet:uri;
              description
                "URI identifying the mount target";
            }
          }
          case host-name {
            leaf hostname {
              type inet:host;
              description
                "Host name of mount target.";
            }
          }
          case node-ID {
            leaf node-info-ref {
              type subtree-ref;
              description
                "Node identified by named subtree.";
            }
          }
          case other {
            leaf opaque-target-ID {
              type string;
              description
                "Catch-all; could be used also for mounting
                 of data nodes that are local.";
            }
          }
        }
      }
    }

    grouping mount-policies {
      description
        "This grouping contains data nodes that allow to configure
         policies associated with mountpoints.";
```

```
      leaf manual-mount {
        type empty;
        description
          "When present, a specified mountpoint is not
           automatically mounted when the mount data node is
           created, but needs to mounted via specific RPC
           invocation.";
      }
      leaf retry-timer {
        type uint16;
        units "seconds";
        description
          "When specified, provides the period after which
           mounting will be automatically reattempted in case of a
           mount status of an unreachable target";
      }
      leaf number-of-retries {
        type uint8;
        description
          "When specified, provides a limit for the number of
           times for which retries will be automatically
           attempted";
      }
    }

    rpc mount {
      description
        "This RPC allows an application or administrative user to
         perform a mount operation.  If successful, it will result in
         the creation of a new mountpoint.";
      input {
        leaf mountpoint-id {
          type string {
            length "1..32";
          }
          description
            "Identifier for the mountpoint to be created.
             The mountpoint-id needs to be unique;
             if the mountpoint-id of an existing mountpoint is
             chosen, an error is returned.";
        }
      }
      output {
        leaf mount-status {
          type mount-status;
          description
            "Indicates if the mount operation was successful.";
        }
```

```
      }
    }
    rpc unmount {
      description
        "This RPC allows an application or administrative user to
         unmount information from a remote datastore.  If successful,
         the corresponding mountpoint will be removed from the
         datastore.";
      input {
        leaf mountpoint-id {
          type string {
            length "1..32";
          }
          description
            "Identifies the mountpoint to be unmounted.";
        }
      }
      output {
        leaf mount-status {
          type mount-status;
          description
            "Indicates if the unmount operation was successful.";
        }
      }
    }
    container mount-server-mgmt {
      if-feature mount-server-mgmt;
      description
        "Contains information associated with managing the
         mountpoints of a datastore.";
      container mountpoints {
        description
          "Keep the mountpoint information consolidated
           in one place.";
        list mountpoint {
          key "mountpoint-id";
          description
            "There can be multiple mountpoints.
             Each mountpoint is represented by its own
             list element.";
          leaf mountpoint-id {
            type string {
              length "1..32";
            }
            description
              "An identifier of the mountpoint.
               RPC operations refer to the mountpoint
               using this identifier.";
```

```
          }
          leaf mountpoint-origin {
            type enumeration {
              enum "client" {
                description
                  "Mountpoint has been supplied and is
                   manually administered by a client";
              }
              enum "auto" {
                description
                  "Mountpoint is automatically
                   administered by the server";
              }
            }
            config false;
            description
              "This describes how the mountpoint came
               into being.";
          }
          leaf subtree-ref {
            type subtree-ref;
            mandatory true;
            description
              "Identifies the root of the subtree in the
               target system that is to be mounted.";
          }
          uses mount-target;
          uses mount-monitor;
          uses mount-policies;
        }
      }
      container global-mount-policies {
        description
          "Provides mount policies applicable for all mountpoints,
           unless overridden for a specific mountpoint.";
        uses mount-policies;
      }
    }
  }

  <CODE ENDS>
```

## 7. Security Considerations

TBD

## 8. Acknowledgements

We wish to acknowledge the helpful contributions, comments, and suggestions that were received from Tony Tkacik, Ambika Tripathy, Robert Varga, Prabhakara Yellai, Shashi Kumar Bansal, Lukas Sedlak, and Benoit Claise.

## 9. References

### 9.1. Normative References

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <http://www.rfc-editor.org/info/rfc2131>.

[RFC2866] Rigney, C., "RADIUS Accounting", RFC 2866, DOI 10.17487/RFC2866, June 2000, <http://www.rfc-editor.org/info/rfc2866>.

[RFC3768] Hinden, R., Ed., "Virtual Router Redundancy Protocol (VRRP)", RFC 3768, DOI 10.17487/RFC3768, April 2004, <http://www.rfc-editor.org/info/rfc3768>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

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

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <http://www.rfc-editor.org/info/rfc6241>.

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

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

### 9.2. Informative References

[I-D.bjorklund-netmod-structural-mount]
Bjorklund, M., "YANG Structural Mount", draft-I-D.bjorklund-netmod-structural-mount-02 (work in progress), February 2016.

[I-D.haas-i2rs-netmod-netconf-requirements]
Haas, J., "I2RS Requirements for Netmod/Netconf", draft-haas-i2rs-netmod-netconf-requirements-01 (work in progress), March 2015.

[I-D.ietf-i2rs-pub-sub-requirements]
Voit, E., Clemm, A., and A. Gonzalez Prieto, "Requirements for subscription to YANG datastores", draft-ietf-i2rs-pub-sub-requirements-05 (work in progress), February 2016.

[I-D.ietf-netconf-restconf]
Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", draft-ietf-netconf-restonf-04 (work in progress), January 2015.

[I-D.ietf-netconf-yang-push]
Clemm, A., Gonzalez Prieto, A., Voit, E., Tripathy, A., and E. Nilsen-Nygaard, "Subscribing to YANG datastore push updates", draft-ietf-netconf-yang-push-02 (work in progress), March 2016.

[I-D.voit-netmod-yang-mount-requirements]
Voit, E., Clemm, A., and S. Mertens, "Requirements for mounting of local and remote YANG subtrees", draft-voit-netmod-yang-mount-requirements-00 (work in progress), March 2016.

## Appendix A. Example

In the following example, we are assuming the use case of a network controller that wants to provide a controller network view to its client applications. This view needs to include network abstractions that are maintained by the controller itself, as well as certain information about network devices where the network abstractions tie in with element-specific information. For this purpose, the network controller leverages the mount capability specified in this document and presents a fictitious Controller Network YANG Module that is depicted in the outlined structure below. The example illustrates how mounted information is leveraged by the mounting datastore to provide an additional level of information that ties together network and device abstractions, which could not be provided otherwise without introducing a (redundant) model to replicate those device abstractions

```
rw controller-network
+-- rw topologies
|   +-- rw topology [topo-id]
|       +-- rw topo-id                node-id
|       +-- rw nodes
|       |   +-- rw node [node-id]
|       |       +-- rw node-id        node-id
|       |       +-- rw supporting-ne  network-element-ref
|       |       +-- rw termination-points
|       |           +-- rw term-point [tp-id]
|       |               +-- tp-id     tp-id
|       |               +-- ifref     mountedIfRef
|       +-- rw links
|           +-- rw link [link-id]
|               +-- rw link-id        link-id
|               +-- rw source         tp-ref
|               +-- rw dest           tp-ref
+-- rw network-elements
    +-- rw network-element [element-id]
        +-- rw element-id             element-id
        +-- rw element-address
        |   +-- ...
        +-- M interfaces
```

The controller network model consists of the following key components:

- A container with a list of topologies. A topology is a graph representation of a network at a particular layer, for example, an IS-IS topology, an overlay topology, or an Openflow topology. Specific topology types can be defined in their own separate YANG

modules that augment the controller network model. Those augmentations are outside the scope of this example

- An inventory of network elements, along with certain information that is mounted from each element. The information that is mounted in this case concerns interface configuration information. For this purpose, each list element that represents a network element contains a corresponding mountpoint. The mountpoint uses as its target the network element address information provided in the same list element

- Each topology in turn contains a container with a list of nodes. A node is a network abstraction of a network device in the topology. A node is hosted on a network element, as indicated by a network-element leafref. This way, the "logical" and "physical" aspects of a node in the network are cleanly separated.

- A node also contains a list of termination points that terminate links. A termination point is implemented on an interface. Therefore, it contains a leafref that references the corresponding interface configuration which is part of the mounted information of a network element. Again, the distinction between termination points and interfaces provides a clean separation between logical concepts at the network topology level and device-specific concepts that are instantiated at the level of a network element. Because the interface information is mounted from a different datastore and therefore occurs at a different level of the containment hierarchy than it would if it were not mounted, it is not possible to use the interface-ref type that is defined in YANG data model for interface management [] to allow the termination point refer to its supporting interface. For this reason, a new type definition "mountedIfRef" is introduced that allows to refer to interface information that is mounted and hence has a different path.

- Finally, a topology also contains a container with a list of links. A link is a network abstraction that connects nodes via node termination points. In the example, directional point-to-point links are depicted in which one node termination point serves as source, another as destination.

The following is a YANG snippet of the module definition which makes use of the mountpoint definition.

```
<CODE BEGINS>
module controller-network {
    namespace "urn:cisco:params:xml:ns:yang:controller-network";
    // example only, replace with IANA namespace when assigned
    prefix cn;
    import mount {
        prefix mnt;
    }
    import interfaces {
        prefix if;
    }
    ...
    typedef mountedIfRef {
        type leafref {
            path "/cn:controller-network/cn:network-elements/"
            +"cn:network-element/cn:interfaces/if:interface/if:name";
            //  cn:interfaces corresponds to the mountpoint
        }
    }
    ...
    list termination-point {
        key "tp-id";
        ...
        leaf ifref {
            type mountedIfRef;
        }
        ...
        list network-element {
            key "element-id";
            leaf element-id {
                type element-ID;
            }
            container element-address {
                ... // choice definition that allows to specify
                // host name,
                // IP addresses, URIs, etc
            }
            mnt:mountpoint "interfaces" {
                mnt:target "./element-address";
                mnt:subtree "/if:interfaces";
            }
        ...
    }
...
<CODE ENDS>
```

Finally, the following contains an XML snippet of instantiated YANG information. We assume three datastores: NE1 and NE2 each have a

datastore (the mount targets) that contains interface configuration data, which is mounted into NC's datastore (the mount client).

Interface information from NE1 datastore:

```
<interfaces>
  <interface>
    <name>fastethernet-1/0</name>
    <name>ethernetCsmacd</type>
    <location>1/0</location>
  </interface>
  <interface>
    <name>fastethernet-1/1</name>
    <name>ethernetCsmacd</type>
    <location>1/1</location>
  </interface>
<interfaces>
```

Interface information from NE2 datastore:

```
<interfaces>
  <interface>
    <name>fastethernet-1/0</name>
    <name>ethernetCsmacd</type>
    <location>1/0</location>
  </interface>
  <interface>
    <name>fastethernet-1/2</name>
    <name>ethernetCsmacd</type>
    <location>1/2</location>
  </interface>
<interfaces>
```

NC datastore with mounted interface information from NE1 and NE2:

```
   <controller-network>
     ...
     <network-elements>
       <network-element>
         <element-id>NE1</element-id>
         <element-address> .... </element-address>
         <interfaces>
           <if:interface>
             <if:name>fastethernet-1/0</if:name>
             <if:type>ethernetCsmacd</if:type>
             <if:location>1/0</if:location>
           </if:interface>
           <if:interface>
             <if:name>fastethernet-1/1</if:name>
             <if:type>ethernetCsmacd</if:type>
             <if:location>1/1</if:location>
           </if:interface>
         <interfaces>
       </network-element>
       <network-element>
         <element-id>NE2</element-id>
         <element-address> .... </element-address>
         <interfaces>
           <if:interface>
             <if:name>fastethernet-1/0</if:name>
             <if:type>ethernetCsmacd</if:type>
             <if:location>1/0</if:location>
           </if:interface>
           <if:interface>
             <if:name>fastethernet-1/2</if:name>
             <if:type>ethernetCsmacd</if:type>
             <if:location>1/2</if:location>
           </if:interface>
         <interfaces>
       </network-element>
     </network-elements>
     ...
   </controller-network>
```

## Authors' Addresses

Alexander Clemm
Cisco Systems

EMail: alex@cisco.com

Jan Medved
Cisco Systems

EMail: jmedved@cisco.com

Eric Voit
Cisco Systems

EMail: evoit@cisco.com