Designing ConfD for NCS

ConfD 6.6

March 2, 2018

Designing ConfD for NCS

ConfD 6.6

Publication date March 2, 2018

Table of Contents

1. Designing for Manageability and Automation 1
    1.1. Introduction 1
    1.2. Requirements 1
    1.3. Best Practise 2

Chapter 1. Designing for Manageability and Automation

1.1. Introduction

Most network devices today need to be managed by two very different entities; both by human users typically using the CLI, and by management applications (NMS / OSS / Controller / Orchestrator) through a programmatic interface like NETCONF.

Human users and management applications have very different needs, and these needs must be reflected in the interfaces they are using. For example, when a new device is built, it is common that the CLI is required to have a similar look and feel as some other product by the same vendor. There may be interactive "wizards" that guide the human user. On the other hand, a programmatic interface need predictability and consistency.

1.2. Requirements

There are a couple of high-level requirements that need to be fulfilled in order to make a device manageable over NETCONF:

A. The management application needs access to the complete set of YANG modules and submodules. This set must not have any references to any definition in a module outside this set.

   In ConfD, a YANG module can be exported to some specific northbound protocol, or completely hidden. It is important that the set of modules and submodule that are made available to the management application is the set of modules that are exported through the NETCONF interface.

B. When the device data models are designed, it is important that any configuration data is actually modelled as "config true" data. There should not be any configuration data that can be set only via custom rpc operations. I.e., do not do `rpc set-interface-params`. Instead, model the interface as a normal YANG data node, so that it can be retrieved with `get-config` and modified with `edit-config`.

   The reason for this is that every rpc operation requires custom code in the management application, adding to complexity and making automation more difficult.

C. If possible, operational state should be modelled as "config false" data, not as output to custom rpc operations. I.e., it is better to have a list of available interfaces as "config false" YANG data nodes than a custom `rpc show-interface`.

   The reason for this is that every rpc operation requires custom code in the management application, adding to complexity and making automation more difficult.

D. When designing and implementing data models, avoid situations where the server creates data that the client didn't send. For example, in some systems, the client creates some object without specifying the key. When the object is created, the server creates the key (e.g., an UUID) of the object and returns it to the client.

   It is convenient for an operator to not have to fill in a boring and essentially random UUID. Similarly, many programmers find it convenient to not have to specify the UUID of the object they are creating.

The problem here is that this prevents transactional behavior, making automation more complex and error-prone. Consider this example:

The orchestrator needs to create two objects, A and B, each identified by an UUID. B needs to reference A. With automatically generated UUIDs, this cannot be done in a single transaction. The orchestrator would need to first create A, have the UUID returned, then create B with the UUID reference to A. This needs special code in the orchestrator, adding to the complexity. It also requires more round-trips, affecting the performance of the system. Finally it makes error-handling more complex, since if the second operation fails, the client needs to figure out how to undo the first operation.

Some systems invented some special mechanisms to allow the orchestrator to create both A and B in a single operation using some kind of temporary identities. This may work across two tables, but not across subsystems (e.g., B is a Linux process), and certainly not across multiple systems (e.g., B is a different device). It also means that the orchestrator gets out-of-sync with the device, and needs some mechanism to retrieve the generated identities, correlate them with the temporary ones, and update its internal structures.

E. When designing and implementing data models, avoid situations where the server transforms the data written by the client to something else (except when transforming data types to their canonical format, e.g., changing "2001:dB8:0:0:0::1" to "2001:db8::1").

F. NETCONF has the advantage over traditional simple CLIs and SNMP of supporting (large) atomic transactions. This means that the client can send an arbitrary configuration change request (e.g., `edit-config`), and the server will either apply everything or nothing.

This property makes automation much easier for two reasons:

First, it greatly simplifies error handling. If a large configuration change fails in the middle in a non-transactional system, the client needs to figure out a way to back out the changes that actually took place. While this is going on, traffic might be affected, since the server have applied parts of a larger configuration change.

Second, the client can send all changes in one go to the server without having to worry about reordering statements to that if A has a reference to B, B must be created before A. Any system that requires data to be modified in a specific order puts a huge burden on the automation client, which needs to be clever and understand all possible dependencies, and reorder data accordingly. This is error-prone and adds complexity.

It is thus important to not design the system so that it is dependent on which order data is created or deleted within a single transaction.

G. Remember that the set of YANG models visible through NETCONF together make up a programmatic API to manage the device. The models need to be treated as any other API, i.e., properly documented and version controlled.

1.3. Best Practise

The following is a list of ConfD specific recommendations for achieving the requirements outlined above.

1. It is recommended to enable RFC 6022 - YANG Module for NETCONF Monitoring, and also store all YANG modules and submodules in ConfD's load path. This makes it possible for a NETCONF client to retrieve all YANG modules from the device itself.

2. Use `tailf:export` to limit the visibility of modules, rather than the `confdc` command line flag `--export`. The reason for this is that it makes all information available in the YANG module itself.

If some models are special for some protocol (e.g., SNMP, CLI), make use of tailf:export. However, make sure that there are no references out of these sets of modules; each set must be self-contained.

3. If a data model is not exported to NETCONF, make sure it is not imported by some other data model that is exported to NETCONF.

4. Do not use the deprecated statement `tailf:symlink`. Use `tailf:link`. The reason for this is that if the symlink target hidden or in a module that is not exported through NETCONF, the management application does not know that data model for the symlink tree.

5. Avoid using hooks and transforms where both the source and target nodes are visible through the NETCONF interface.

   If hooks or transforms are needed in order to get a desired CLI behavior, use them in a data model exported specifically to the CLI.

6. Never validate configuration changes; instead always validate the configuration result.

   For example, do not add code that makes it impossible to change the configuration of some object without first disabling it. The reason for this is that automation becomes impossible without special code, and it breaks the transactionality of the system. If the application wants to change this piece of configuration, it needs to first figure out that some other object needs to be set to 'disable' (how does it know this in the generic case?), and then commit, make the change, commit again, and then set to 'enable' (thus splitting the single transaction into three).

7. Be careful when you update your YANG models. Follow the update rules defined in Section 10 of RFC 6020 as much as possible.