Automation Principles for Designing Network Elements

tail-f a Cisco company

Table of Contents

Automation Principles for Designing Network Elements

1. Introduction

In network management, automation is defined as the set of technologies and their implementation that allow for management operations to be automated with minimal human interaction. One type of management operation is the configuration of network elements by a controller or orchestrator.

Networks consist typically of a heterogeneous group of network elements (both physical and virtual) which expose different APIs for management purposes. When designing network elements, it is important to take into the consideration how a network element can best provide support for automation. This document discusses how to implement best practice automation support in network elements using NETCONF and YANG.

2. Definitions

2.1 Programmability

Programmability is the ability of a network element to be configured (programmed) using APIs. A good programmatic API must verify the following:

- A well-defined set of operations.
- Strong input data definition, not leaving any doubt as to what data type or constraints need to be verified before using the data.

Standards based programmatic APIs such as NETCONF and YANG provide a more advanced feature rich API than other programmatic APIs for network management. NETCONF defines the operations and YANG defines data and the relationship and constraints it has for a coherent (valid) configuration.

2.2 Automation

Automation plays the biggest role in driving requirements for today's network management solutions, whether it is at a network element (device) level or at the orchestration level (Orchestrators sometimes are themselves managed by other orchestrators). There are several levels of automation that can be achieved starting from non-standard based scripting of management operations to standards-based APIs that abstract device details and create a common denominator for all management operations regardless of the device types and vendors. This kind of automation requires a strict design of the APIs exposed northbound of network elements, controllers, and orchestrators. This strict design is partially covered by the standardization effort via standard RFCs, but there is more to it that remains the decision of the API implementors. A simple example is NETCONF support. This support can be compliant with the RFCs while at the same time can be broken for the advanced automation needs. As explained later in this document, there are ways to get things wrong and complex for automation, but fortunately there are ways to get things right as well. We will explore some of the common issues and the best practices for automation when implementing a NETCONF and YANG API at the network element level using ConfD.

2.3 Configuration Side Effects

Configuration side effects refers to the extra configuration that a network element creates and adds in addition to the user configuration set via an <edit-config> or <edit-data> RPC in NETCONF. This extra configuration may or may not affect automation, depending on its exposure to automation tools.

2.4 Auto Configuration

Auto configuration refers to configuration that is created by a network element which isn't a direct result of a northbound management interface operation such as an <edit-config> or an <edit-data> RPC. Auto configuration is more general than configuration side effects because it can happen outside of incoming northbound management transactions.

2.5 Out-of-Band Configuration

Out-of-band configuration refers to any configuration set from other northbound management interfaces such as CLI. Out-of-band configuration can affect automation if it modifies the same configuration that is managed by automation tools.

3. Automation Principles, Potential Issues, and Best Practices

This section discusses the requirements that controllers and orchestrators have towards network elements in order to enable high levels of automation and potential issues with some of the NETCONF implementations that exist in the field.

3.1 Programmatic API: YANG Access

As with any API, the user of the API needs access to the operations as well as to the data that the operations act on.

In network management, the controller or orchestrator needs access to the complete set of YANG modules and submodules needed by higher level services. This set must not have any references to any definition in a YANG module outside of this set.

Access to the YANG data models supported by network elements is mandatory for management applications to understand the relationship between different data nodes and the constraints that need to be enforced during configuration. This eliminates unnecessary configuration errors, removes the need to implement device specific behavior at the controller or orchestrator level, and enables efficient configuration across the network. Such access is best provided by supporting the NETCONF RPC <get-schema>. A network element should support the ietf-netconf-monitoring.yang module which defines the <get-schema> NETCONF operation that allows a controller or orchestrator to retrieve schema files (YANG modules) from the network elements.

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 submodules that are made available to the controller or orchestrator is the set of modules that are exported through the NETCONF interface.

3.1.1 ConfD Best Practice

- It is recommended to enable RFC 6022 - YANG Module for Netconf Monitoring.
- Use the "tailf:export" extension to limit the visibility of modules, instead of the confdc

command line flag "--export". The reason for this is that it makes all information available in the YANG module itself. If some YANG modules are specific to some non-NETCONF northbound interfaces, such as SNMP and CLI, make use of the "tailf:export" extension to exclude these modules. However, make sure that there are no references which point outside of these sets of modules; each set must be self-contained. In addition, make use of YANG annotation modules to add these "tailf:export" statements, so the original module remains untouched. This helps with future upgrades and reduces potential errors.

- If a YANG module is not exported to NETCONF, make sure it is not imported by some other YANG module that is exported to NETCONF.

3.2 Configuration Data Modelled as "config true"

When modeling data in YANG, special attention needs to be paid to the type of data being modelled, whether it should be configuration or operational data. While this is obvious as a first step, it still remains a source of problems in the field.

Configuring a network element should always be possible using the standard NETCONF operation <edit-config>. Introducing any exception to this results in the network element becoming difficult to automate. An example of this is the introduction of a custom RPC that must be used (as a hard requirement introduced by the network element implementation) to change configuration. This custom RPC is only acceptable if controllers and orchestrators can still rely on the standard way of changing configuration to achieve the same desired goal. Controllers and orchestrators should not be required to know, and implement, a custom RPC needs to be used to configure the network element.

Modeling configuration data as "config true" and not as "config false" allows for more protocol level features that are specific to configuration data. Example features are:

- Datastore used: Configuration data is stored in the Running datastore which constitutes the source of truth for configuration at the network element level.

- Operations: Copying configuration data to preserve the configuration across devices during maintenance for example. Standard <copy-config> only applies to configuration data and cannot be used for state (operational) data. The NETCONF ":validate" capability applies to configuration data alone.

In summary, configuration data needs to be modelled as "config true" data and the NETCONF implementation needs to support <edit-config> and <get-config> for such data and not expect the NETCONF client to use custom RPCs to manage this type of data. Adding such complexity to controller and orchestrator implementations makes automation more difficult.

3.3 Configuration Data Modelled as "configf false"

The differentiation between what is operational and what is configuration data, at the YANG level, is there for a reason. Mixing up the two by modeling configuration data as "config false" or operational data as "config true" will break programmability. A controller or orchestrator should be able to request configuration of a device using <get-config>. The server in this case will only return data that is modelled as "config true". If operational data is also modelled as "config true", this will be considered configuration and no longer operational data. If some data is modelled as configuration while it should be operational, errors will arise later such as when trying to copy configuration from one network element to another.

Other problems can happen at the automation level when the network element state changes

in a way that does not match the wrongfully modelled configuration data, hence resulting in an out of synch situation between the network element view of the data and the controller or orchestrator view.

Another requirement here is to model operational data as "config false" and not as RPC output data. This enables the possibility of storing operational data in the operational data store as defined by NMDA.

The challenge of using custom RPCs to output operational data is similar to the challenge of using custom RPCs to configure a device: controllers and orchestrators need to add custom code to handle such operations, resulting in unnecessary complexity at the automation level. Closed-loop systems rely on standard RPCs to consume operational data and act on any potential network events.

3.4 Side Effects

Side effects of configuration happen when a network element adds more configuration than what was sent by a controller or orchestrator. These are called side effects because after a manager commits a certain configuration and does a <get-config> to see what was committed, it will find out that there are more changes in the network element than what were requested.

An example of a side effect is when the controller or orchestrator creates some object without specifying the key. When the object is created, the network element automatically creates the key (e.g., a UUID) of the object. While it is convenient for an operator to not have to fill in a boring and essentially random UUID, it causes the controller or orchestrator's view of the configuration to become out of sync. This convenience should be handled by the controller or orchestrator and not the network element. For example, by having a WebUI front end pre-fill a UUID before the configuration request is sent to the device.

Consider this more complex example: The orchestrator needs to create two objects, A and B, each identified by an UUID. B needs to reference A. With network element generated UUIDs, this cannot be done in a single transaction. The orchestrator would need to first create A, read back the UUID, 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 special mechanisms to allow the orchestrator to create both A and B in a single operation using temporary identities. This may work across two tables, but not across subsystems (e.g., A & B are different Linux processes), and certainly not across multiple systems (e.g., A & B are different devices). 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.

3.4.1 ConfD Best Practice

- 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 YANG data model exported specifically to the CLI.

3.5 Data Transformation

This problem is a bit like the side effects problem described above with the small difference that the network element implementation, instead of adding or creating more configuration, it modifies the configuration sent by the client.

An exception to this exists when a client modifies the data to its canonical format. For example, changing "2001:dB8:0:0:0::1# to "2001:db8::1" is fine as the data type is a standard type, known to both the client and the server.

Another occurrence of this issue is when the value set and returned by the device later does not match the data type used in the YANG data model.

3.6 Ordering

The "ordering" problem, sometimes referred to as "sequencing", occurs when the network element requires configuration to be committed in some specific order, meaning some configuration needs to be fully committed in one transaction before the next configuration is committed. This introduces a breach to the transactional behavior that each NETCONF implementation should adhere to. This breach of transactional behavior results in several problems that break automation. The YANG modules, representing a device contract, are no longer sufficient enough for a controller or orchestrator to successfully manage the configuration of a device. In this case, complexity was introduced at the device level requiring controllers and orchestrators to know about the specific order needed.

Error management becomes complex as well. When a certain configuration is pushed by the client in a certain order and split into several transactions, the client needs to track data what needs to be explicitly rolled back in case of errors. Transactions no longer become sufficient to preserve the coherent state of a network element.

The benefit of transactional management, and NETCONF specifically, is lost when the ordering requirement is introduced.

In the NETCONF world, a controller or orchestrator should be able to send configuration data for all the YANG modelled configuration nodes in a single <edit-config> transaction, without specifying order. A network element's NETCONF server needs to be able to accept such configuration based only on the constraints expressed in YANG.

3.6.1 ConfD Best Practice

- Never validate configuration changes; instead, always validate the upcoming configuration. 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 specialized 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?), commit, make the configuration change, commit again, set to 'enable', and commit yet again. Thus, splitting the single transaction into three.

3.7 Aliasing

Aliasing happens when a network element exposes the same configuration data via two different modules or more. This can happen when an implementation needs to support both IETF and other Standards Organization’s data modules, for different customers or applications. Aliasing is a problem when two or more YANG modules, representing the

same data, are exposed to the same management application (orchestrator). In this case, an <edit-config> sent for one of the modules nodes results in the configuration being updated for two modules, even if the storage is one and only one on the device side. A <get-config> operation would return data in at least two different namespaces, resulting in a synchronization issue between the manager and the managed device. This behavior is a bit like side effects in that it creates more data than was originally sent to the device.

A vendor should document which non-overlapping modules can be used for any given use case, until IETF defines a standard way for packaging modules.

Another, more complex, scenario of aliasing is when the overlap of functionality happens between nodes in the same YANG module. This usually happens when deprecating a node and adding a new node (with a different name). Aliasing due to deprecation of nodes is needed sometimes to preserve backward compatibility with older versions of clients and to give time to implementors to make the necessary changes. These deprecated nodes should be ignored by new northbound clients.

3.7.1 ConfD Best Practice

- When multiple views (YANG modules) are used for the same data, make sure only one set is exposed to the manager and not both. The "tailf:export" extension can be used to decide which module to expose to NETCONF clients. When building a NED (Network Element Driver) in NSO, it is possible to manually select which modules to include in the NSO NETCONF NED. It is then not required to use the "tailf:export" extension, as part of building the NED is knowing which YANG modules need to be used. But it is best to not even expose those "duplicate purpose" YANG modules to NETCONF clients.

3.8 Defaults Handling

The handling of default values for YANG models needs to be explicit. A NETCONF server needs to support the ":with-defaults" capability so it can inform the client how default values are handled by the server. It can also be used by the client to control whether default values should be generated to replies from the server or not.

This capability is enabled by default in ConfD.

3.9 Empty Configuration Validity

When modeling the new cool feature that everyone wants to use, it may be hard to imagine that it may be included in systems where the user might not be interested in your module—at least not right now.

If you are adding a new feature in your product that involves new YANG modules or new YANG changes such as a new leaf or list, it may be hard for some users to come up to speed on these latest changes and upgrade their tooling promptly. It is therefore wise to ensure that an empty configuration remains meaningful and allowed in the new YANG schema.

This means your changes to the data model, along with all the syntactic and semantic rules you added, should accept empty configuration. This way, when upgrading to the new YANG data model, previously used tools remain operational and stored configuration remains valid.

For example, if you are adding BGP support in your data model and knowing that BGP can only work if an Autonomous System (AS) number is configured, make sure that the AS number is only required if BGP is used.

A common way to achieve this is to use a presence container, as shown in the following example. This allows the user to enable and disable the entire function and to only require any mandatory elements if the function is in use.

```
container bgp {
  presence "Enable BGP";
  leaf as-number {
     type uint32;
     mandatory true;
  }
```

3.10 Configuration Dependency on Operational Data

When configuration is made dependent on operational data (device state), automation becomes difficult because a once valid configuration isn't always guaranteed to be valid by the NETCONF implementation on the device. This is known as operational state dependent configuration in network management theory and is considered to be a very bad thing. If operational state dependent configuration is used, it means that if the management application wants to bring back a device to an already known-to-be-valid state, it cannot rely simply on the configuration used. There can be manual intervention needed by a human user which breaks automation in addition to a device now not only depending on configuration to move from one state to another but also depending on current operational state.

An example of this is how certain constraints are expressed in YANG. YANG "must" statements use XPath expressions that can be used to constrain values to an acceptable set. This is a powerful mechanism whereby it is possible to instruct ConfD to compute an XPath expression whenever a configuration change is attempted. This makes it possible to have value constraints that depend on other parts of the configuration. However, the constraints must only refer to other parts of the configuration and not operational data as per RFC 6020 and RFC 7950. The violation of this will result in errors when compiling the YANG modules or when building the NETCONF NED using the NETCONF NED Builder. Having this dependency in your YANG data model will cause interoperability issues with Cisco NSO and other service orchestrators.

In ConfD, it is possible to implement validation logic outside of YANG modules. This is a good practice in general, to be able to add complex validation rules that are based on anything, but it doesn't prevent a user from validating configuration data based on a certain state. This is why it is easy to introduce this problem in ConfD clients (southbound integration).

3.10.1 General Best Practice

When implementing validation checks of configuration data, whether in YANG itself or outside (such as in validation callbacks), make sure that configuration that is considered (as a whole) valid one time is always valid other times.

Wholesomeness here means a snapshot of the entire device configuration, if copied and sent to another device (with the same implementation), regardless of the device state, is a valid configuration.

In some implementations, the operational state of a device such as the presence of hardware can dictate whether a configuration is valid or not. In these cases, instead of rejecting configuration, it should be accepted and an "operational state = down" field should be used to declare the configuration inactive. When the condition is verified later, such as the insertion of a line card in a chassis system, the configuration is already present, the implementation can then toggle the operational state field to "up".

3.11 Transaction Issue Scenarios

3.11.1 Transaction Fails but Some of the Configuration was Modified on the Device Anyway

The atomicity property of transactions is violated in this case. When a transaction fails, all configuration changes within that transaction should be reverted. ConfD takes care of this as long as the device that participates in the transaction during the validate or prepare-phase can handle this properly. This means a device shouldn't apply configuration while it's still being validated or after the validation phase, in the prepare-phase, without implementing the proper abort callback in case the transaction fails later.

3.11.2 Transaction Fails Due to Certain Previously Configured Values Not Included in This Transaction

When a network element validates configuration, it should validate the entire resulting configuration and not just look at what has been changed in a transaction. This means the whole configuration should be considered and not only what is part of the transaction as transactions can target a small subset of the overall configuration. In other words, validate the configuration as a set and do not analyze individual changes. Doing so results in fewer lines of code.

3.11.3 Transaction Fails Due to Dependency Between Subsystems in a Single Transaction

Another of saying this is you can't configure both A and B in the same transaction. This issue results in an ordering issue although not as explicit. When there is such a requirement, this forces the northbound controller or orchestrator to learn about the specific behavior of the device and it cannot rely on the contract which is the set of YANG modules exported to the NETCONF clients.

This requirement can be handled by abstracting the device logic from the management logic so that A and B can still be configured in the same transaction. This is accomplished by the device itself implementing the order of application of configuration changes. In ConfD, this is achieved using CDB subscription priorities.

3.11.4 Transaction Fails Due to the Dependency on a Prior Configuration

Such implementation breaks consistency with regards to valid configuration. A manager can't go back to a once valid configuration. An example of this is element A is not mandatory, but once created, it can't be deleted.

Let's assume we have a configuration represented by "C-A" (C minus A), that is valid. Once we add A, it becomes C, which is also valid. A northbound management application or automation tools can't go back to C-A because the network element rejects this change. This breaks automation. What is validated in this case is not data itself but the operation of removing A. This is no longer data model driven management as validation in data model driven management focuses on the resulting data set and not the individual changes after a transaction is committed.

3.11.5 Transaction Fails Due to Dependency on Operational State

For example, element A can't be configured unless a certain piece of hardware is present. This is an example of operational state dependent configuration. In this case, the presence of the hardware affects whether configuration transactions succeed or fail. This has a negative effect automation and makes it less deterministic.

Configuration validation in transactions shouldn't depend on hardware presence or other operational state changes.

3.11.6 There was no Transaction, but the Configuration Changed

This is system driven automatic configuration and not a northbound manager driven configuration. This, of course can, be fine if the changed configuration isn't exposed northbound. Otherwise, this causes an out-of-sync scenario with controllers and orchestrators and has a negative effect on automation.

3.12 Backward Compatibility

It is extremely important to consider the backward compatibility impact on existing tools and configuration when making changes to a YANG data model. Failing to do so can bring down an entire network service by rendering some network elements un-operational, leading to hours, and even days, of debugging.

In some cases, breaking backward compatibility is acceptable and that's typically when a node in YANG was never used or was experimental and was meant to be removed. This requires an intimate knowledge of how YANG modules are being consumed by the end customers.

The YANG RFC 7950 talks about backward compatibility and how to update YANG modules. RFC 8407, a guideline for module authors and reviewers, also lists guidelines to improve the readability and the interoperability of YANG modules. These guidelines should be followed.

4. Summary

This application note has taken a look at what is needed for network elements to be easily and properly incorporated into network automation. By following the guidelines provided, the network element which you are designing will be poised for success in the modern world of network automation.

5. For More Information

For more information about ConfD,visit https://www.tail-f.com

For more information about NETCONF & YANG Automation Testing (NYAT), visit https://info.tail-f.com/netconf_yang_automation_testing

To learn more about transactions andnetwork wide transactions, read the whitepaper at https://info.tail-f.com/managing-distributed-systems-using-netconf-and-restconf

tail-f a Cisco company

www.tail-f.com
info@tail-f.com

Corporate Headquarters

Sveavagen 25
111 34 Stockholm
Sweden
+46 8 21 37 40