Device Transactionality

tail-f a Cisco company

Table of Contents

Device Transactionality

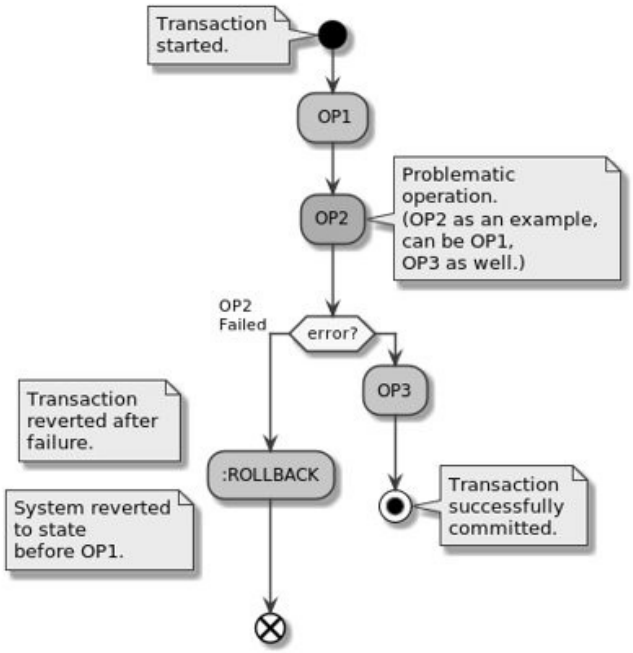
1. Introduction

This application note is about device transactionality in the world of modern network and configuration management. Transactionality plays a key role for automation and programmability of network devices. Standard transactional protocols were introduced (e.g., NETCONF) to support transactionality on the devices. Transactional devices can be easily plugged into network management systems (orchestrators). This is important for automated network configuration and transactionality. Many existing devices (e.g., legacy devices) do not support transactionality but can be made so by implementing a transactionality adaptation layer.

In this application note, we review basic transactionality principles and then describe differences between transactional and non-transactional devices as well as discuss common transactionality issues. Then, we will take a look at an adaptation layer that can "upgrade" a device to be transactional. To wrap up, we present a ConfD example application that implements such an adaptation layer.

2. Transactionality Principles

Basic transactionality concepts are probably well known to you. All operations in the transaction are treated in an atomic way. This means, all operations complete successfully and, if not, nothing will be changed in the system the transaction is running on. The system remains in its original state. Transactionality also simplifies implementation of rollback support.



2.1 ACID

A transaction can be seen as independent unit of work with ACID properties.

- **Atomicity** - All operations within transaction are treated as single unit.
- **Consistency** - Data is in a consistent state, i.e., all (data model) constraints are fulfilled before the transaction starts and after the transaction finishes.
- **Isolation** - Concurrent transactions may be running and they do not influence each other.
- **Durability** - If the transaction was successfully completed, the changes persist. This requires transaction to be recorded in some permanent memory.

2.2 Network Wide Transaction

A network consists of many devices and is usually managed by a network orchestrator such as Cisco NSO. Configuration of all devices within the network is called "Network Wide Configuration". The full network can be transactional. In this case, all devices must ensure some level of transactionality (native or added by some layer). A transaction over a Network Wide Configuration is called a "Network Wide Transaction".

3. Transactionality and Devices

Let's look at transactionality in the scope of devices and implementation of a Network Management Protocol.

3.1 Devices with Transactional Network Management Protocol

- Support for transactions (e.g., NETCONF protocol with `commit` and `confirmed-commit` capabilities)
- Simple integration into network management systems
- Support for programmability and automation
- Rollback is usually supported directly or is simpler to perform (e.g., by network management system)

3.2 Devices with Non-Transactional Network Management Protocol

- Transactions are not supported (e.g., SNMP, proprietary non-transactional CLI)
- When configuring, one operation can bring a device into failed state
- Special knowledge of required command order and rules is needed
- Usually requires human operator with detailed knowledge of device behavior
- Cannot be easily automated without additional layer
- No simple rollback
- More expensive to operate
- May be cheaper to develop (such devices may evolve from successful proof of concept

4. Device Adaptation

Transactional devices have an advantage over non-transactional devices because they are more suitable for automation. There are wide range of devices, that do not support transactionality (legacy, proof of concepts, etc.). Very often they operate fine and may be performant but are not reliable from configuration point of view.

Non-transactional devices themselves can be made transactional with adaptation.

- Non-standard adaptation - Uses a proprietary transactional protocol
- Standard adaptation - Uses a standard transactional protocol (e.g., NETCONF) and can be easily added to existing network orchestrator

The adaptation layer can either be a proprietary application or it can use an existing management agent framework (e.g., Tail-f/Cisco ConfD) with API based adaptation.

Part of the adaptation task is to identify device issues that break transactionality and to figure out how to fix them. For example, a non-transactional device may have configuration elements which can be configured only in a specific order. However, in a transaction, the order of changes is not relevant. Therefore, this results in transactional rules which the adaptation layer must handle, i.e., send commands to the device in the correct order.

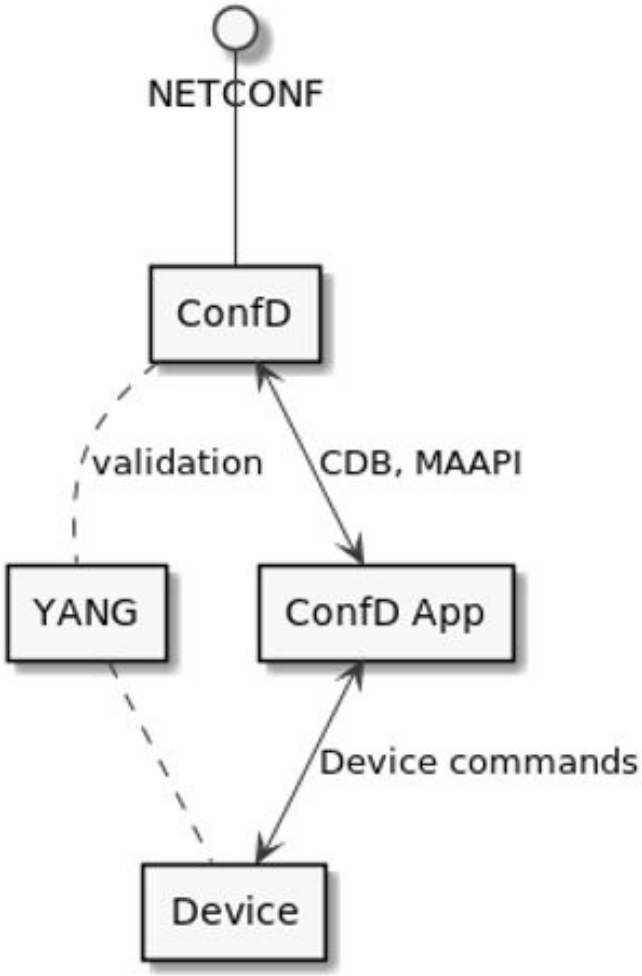
The following is the list of the most common issues that need to be addressed by an adaptation layer:

- Transaction fails, some configuration was changed anyway.
- Transaction fails due to certain previously configured elements not included in this particular transaction.
- Transaction fails due to dependency between elements in single transaction, e.g., you can't configure both A and B in the same transaction or you must configure elements in a certain order (e.g., create A before B, delete B before A, etc.)
- Transaction fails due to the dependency on a prior configuration, e.g., element A isn't mandatory, but once created, you can't delete it
- Transaction fails due to dependency on operational state, e.g., you can't configure A while the operational state of a component is down.
- Transaction succeeded, but the representation is different ("autoconfig")
- There was no transaction at all, but the configuration changed spontaneously anyway.

5. ConfD Based Device Adaptation

ConfD is a data model driven configuration management plane framework that provides a programmable API which can be used to simplify adaptation of non-transactional devices. ConfD supports transactions and all configuration data is stored in the database (called CDB), which is updated in a transactional manner.

Adaptation is done in the form of a ConfD application that connects to ConfD. The adapted device can use ConfD to support the NETCONF protocol with commit or confirmed-commit transactional operations. Such a device can be easily integrated with a network orchestrator such as Cisco NSO.



The approach to solve the problem (add transactionality) usually consists of the following items:

- Identify transactionality issues for the device
- Create YANG data model, add YANG validation constructs
- Implement required subscription and validation callbacks

The Implementation itself usually consists of the following tasks:

Validation to ensure the data model consistency via:

- YANG constructs - e.g., `leafref, pattern, range, must, when` (XPATH expressions in the constructs should be designed carefully so the complexity can be handled by XPATH evaluation engine).

- Validation points with validation implemented in code in a ConfD validator application.

CDB Subscriptions with priorities to ensure the correct command order.

Note: There may be more additional advanced implementation tasks like data provider callbacks, transformations, etc.

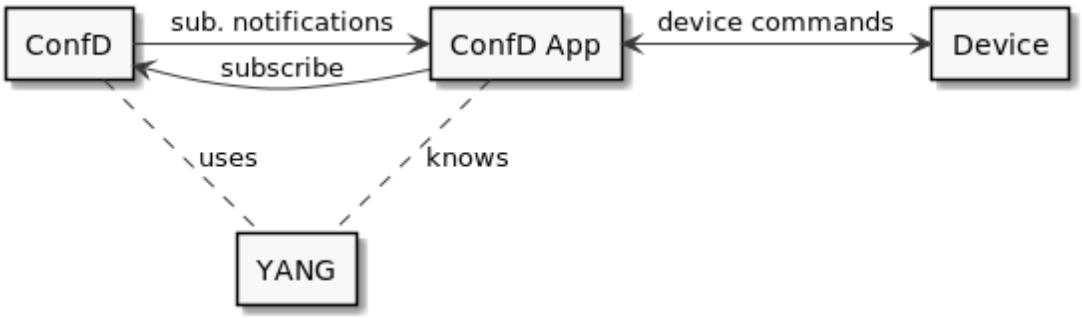
5.1 CDB Subscriptions

CDB Subscriptions are a powerful ConfD mechanism which is used to notify a ConfD subscriber application (transactionality layer) about changes in the configuration. Ability to group subscriptions according to priority is a key feature for transactionality adaptation.

The order in which the configuration elements are populated in a transaction is irrelevant. However, non-transactional devices very often require configuration commands to be sent in specific order. This can be handled by subscription priorities. When a configuration change is received by ConfD and written to CDB, the subscription notifications are sent in priority order and are converted to device specific commands by the subscription handling application.

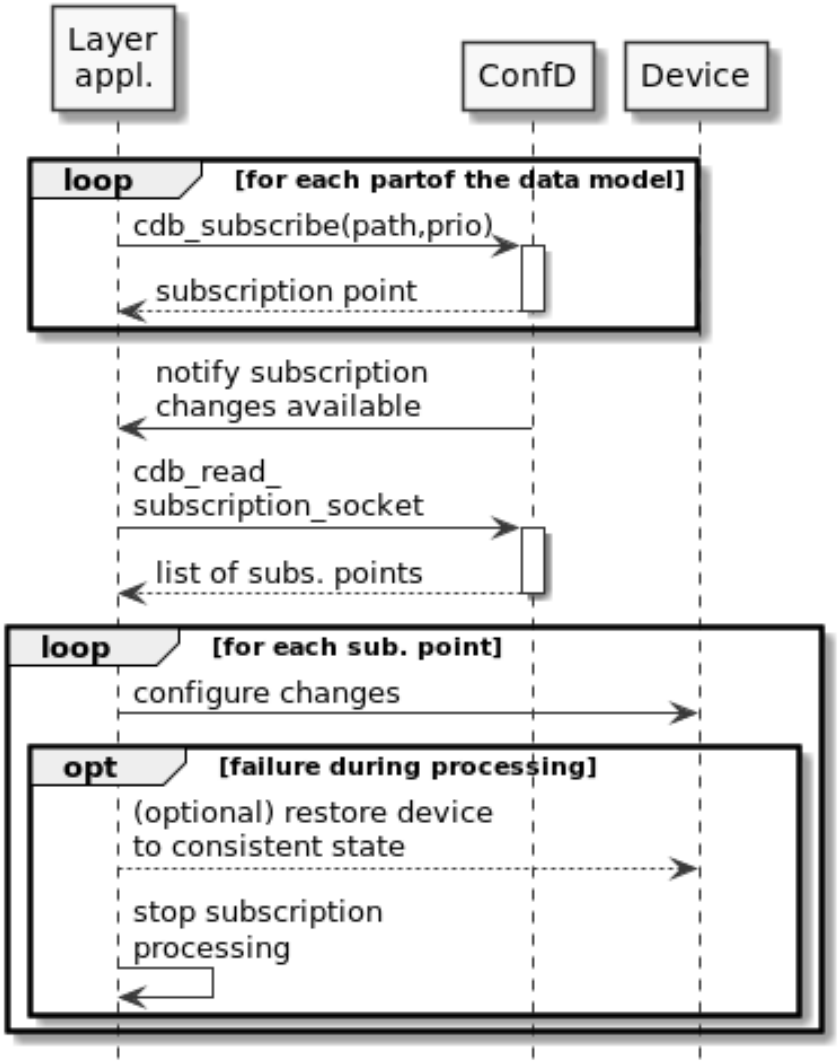
A ConfD subscriber application subscribes to parts of the YANG data model. ConfD sends data change notifications according to the requested subscription priorities.

There may be several subscribers (even across more than one application) with different priorities. This can reduce application complexity as it does not have to handle caching and sorting of the notification messages.



A typical subscription scenario:

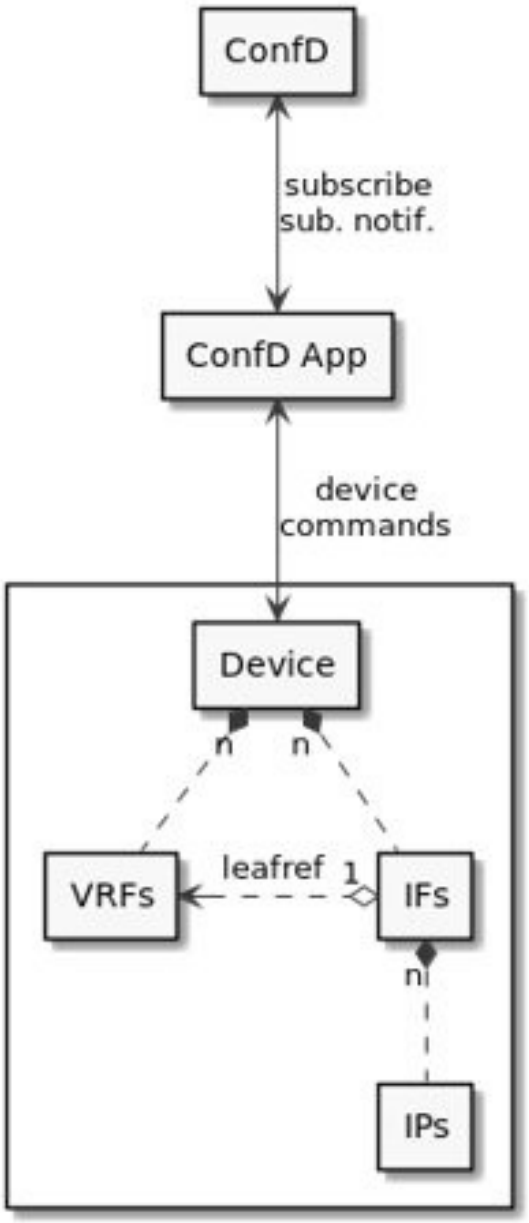
- Configuration data is stored in the ConfD CDB.
- Subscriptions monitor changes in the selected parts of the data model.
- Several subscriptions with different priorities can be created.
- Subscriber application receives notifications about changes in order of the specified priorities.
- Subscriber application propagates changes to the device.
- In case of failure, there is mismatch between what is in the CDB and what is configured. This is an exceptional situation since configuration is validated by YANG constraints and validation points. Already processed subscription points can be handled and device can be put into a consistent state. Operator should be notified.



Note: An in depth CDB API description can be found in the ConfD User Guide.

6. ConfD Example

In the ConfD distribution, there is an example application which can be found in the examples.confd/nso_interop/transactionality directory.
This is an example transactionality layer that is implemented as a ConfD application and works with a hypothetical device with interfaces and VRFs. The device can be configured with text-based commands, e.g., CLI commands. For this hypothetical device, it is important to follow rules (i.e., command order and restrictions), otherwise the configuration will not be successful.



For this example, the identified transactionality rules for the device are:

1. IP address(es) can be assigned only to the configured interface.
2. Only interface(s) without an IP address can be deleted.
3. Only existing VRF(s) can be assigned to an interface.
4. A VRF can be deleted only if it is not assigned to an interface.

Rules are enforced with YANG leafref and subscription priorities. The text commands are sent in the following order (from high to low):

- Create VRF(s) - rule 3.
- Delete IP address(es) from the interface(s) - rule 2.
- Delete, create (add), change interface(s) - rules 2 & 3.
- Create (add) change IP address(es) - rule 1.
- Delete VRF(s) - rule 4.

6.1 Example YANG Data Model

This is the tree representation of the example's YANG data model:

```
+--rw sys
     +--rw vrf* [name]
     |  +--rw name    string
     +--rw ifc* [name]
        +--rw name           interface-name
        +--rw description?   string
        +--rw enabled?       boolean
        +--rw hw
        |  +--rw speed?    interface-speed
        |  +--rw duplex?   interface-duplex
        |  +--rw mtu?      mtu-size
        |  +--rw mac?      string
        +--rw vrf?           -> /sys/vrf/name
        +--rw ip* [address]
           +--rw address          inet:ipv4-address
           +--rw prefix-length    prefix-length-ipv4
           +--rw broadcast?       inet:ipv4-address
```

See: `example.yang`

```
leaf vrf {
     type leafref {
        path /sys/vrf/name;
     }
}
```

As you can see, there is `leafref` reference from the interface element to the VRF list. This ensures the VRF assigned to the interface exists. It also ensures that a VRF can be deleted only if not assigned to an interface.

6.2 CDB Subscriptions Priorities

Subscriptions are sorted according to the device transactionality rules (see: cdb_client.c).

```
subs[] =
   {{.subpath=vrf_subpath, .iter=vrf_iter},
   {.subpath=if_subpath, .iter=handle_delete},
   {.subpath=if_subpath, .iter=if_iter},
   {.subpath=ip_subpath, .iter=ip_iter},
   {.subpath=vrf_subpath, .iter=vrf_delete}};
…
 /* use priorities 100, 200, 300, ... to allow other
    subscribers to squeeze in */
OK(cdb_subscribe(ss, (i+1)*100, 0,
                 &subs[i].subid,subs[i].subpath));
```

Priorities are set according to the order in the array (first element - highest priority) (see cdb_client.c.).

To enforce the rules, we specify subscription priorities in a way such that the notifications for data model paths and operations like delete, create, modify, come in required order:

- first, we get notification for VRF creation (as it is not dependent on the other parts, and it ensures rule that VRF must exist, before it can be assigned to the interface)
- next, we process deletion of IP addresses under interfaces, this ensures rule that interface can be deleted only if all IP addresses are removed
- after that, we process creation, deletion, or modification of interfaces, in case of delete we can be sure there are no IP addresses, in case of VRF assignment we know VRF has been already created
- next, creation and modification of IP addresses is handled. In case of addition, we know interface is already configured
- finally, we handle delete of VRF. At this point we know, that VRF reference was removed from the interface in previous notifications.

6.3 Example Demo

In the example directory, you will find a README file, describing the example, how to build it, and how to run it. It also describes steps to demonstrate how the transactionality layer works.

The device text commands are printed to standard output and one can see from the output that order of the commands reflect device transactionality rules. If we try to make a step that breaks transactionality (e.g., delete a VRF that is assigned to an interface), the transaction is rejected. Operator does not need to remember device rules and can configure the device data model in an arbitrary order. When a transaction starts, the adaptation layer correctly solves command order of the configuration.

6.4 Limitations

The example is simple and, as a result, there are limitations.

There is no real connection to a device; device commands are only emulated and printed.

We assume subscription processing cannot fail and only use one-phase subscriptions. In the real word, this is not always possible (e.g., network disconnection, power outage) and there is not a 100% non-failing solution.

Using the ConfD CDB API you can add one more layer of error handling in form of two-phase subscription. See the CDB API documentation in the ConfD User Guide for more information.

More information can also be found at https://www.tail-f.com/resources/, including other whitepapers, application notes, and videos.

7. Summary

It is possible to use a management plane framework such as ConfD to adapt non-transactional devices to be transactional with full support of a NETCONF interface. Key ConfD features to support this are validation and CDB subscriptions.

Such adapted devices can be easily plugged into a network orchestrators' (like Cisco NSO) ecosystem. Thanks to NETCONF, there is no need to program an orchestrator specific adaptation layer.

NETCONF support simplifies automation and programmability and supplies a standard transactional configuration management protocol.

An adapter as described in this application note can be implemented using the free ConfD Basic. Additionally, a great way to test an adapter is to use NETCONF & YANG Automation Testing (NYAT).

8. For More Information

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

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

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

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

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