

# ConfD EVALUATION KICK START GUIDE

On-device management framework for  
Network Equipment Providers (NEPs)





# Table of Contents

<b>Getting Started with ConfD .....</b>	<b>3</b>
Installation .....	3
Setting up Your Environment.....	4
Finding Documentation .....	4
Launching a First Example .....	4
Watching a Canned Demo .....	4
<b>ConfD Anatomy &amp; Architecture .....</b>	<b>5</b>
Management Interfaces.....	6
YANG Modeling.....	7
Brief Transaction Theory.....	8
The Core Engine and CDB Database.....	10
Application APIs .....	12
High-Availability .....	15
SNMP Considerations.....	15
<b>Running ConfD.....</b>	<b>16</b>
Data-Model Definition .....	16
Data-Model Compilation .....	19
Rendered Management Interfaces.....	19
Configuration Data-Store, Rollbacks.....	21
Making the Configuration Changes Happen .....	22
Operational Data Providers.....	25
Configuration Validation.....	29
A System Example.....	30
<b>The Bigger Picture of Network Management .....</b>	<b>30</b>
The Operator View.....	30
FCAPS .....	31
Network Automation and Software Defined Networking.....	31
NCS .....	32
<b>Evaluation Checklist .....</b>	<b>33</b>
Summary.....	35
<b>Further Information .....</b>	<b>35</b>

# GETTING STARTED WITH CONFD

Welcome to the world of ConfD on-device Configuration Management. This Guide's aim is to quickly bring you up to speed with ConfD, both on a hands-on level as well as understanding its architectural principles and how it fits in with the surrounding world.

Readers interested in understanding the what and why about ConfD can skip directly to the ConfD Anatomy & Architecture chapter (page 5).

## Installation

ConfD is delivered as a self-extract archive, which is OS/CPU specific, plus a number of generic tar-archives with documentation, examples, etc. When you run the installation, the self-extract archive will unpack both itself and the tar-archives, provided that they are located in the same directory.

ConfD can be installed either locally in your home directory, or in a central location for multiple users to use.

To install ConfD, just run the installer with a single argument, the directory where you would like ConfD to be installed. The installation directory you specify must either not yet exist, or be empty. No files will be added or modified outside the directory you specify.

```
$ sh confd-<VERSION>.<OS>.<ARCH>.installer.bin /path/to/confd
```

On a typical system, you would see something like this:

```
$ sh download/confd-X.X.linux.x86_64.installer.bin ~/confd-X.X
INFO Unpacked confd-X.X in /home/someuser/confd-X.X
INFO Found and unpacked corresponding DOCUMENTATION_PACKAGE
INFO Found and unpacked corresponding EXAMPLE_PACKAGE
INFO Generating default SSH hostkey (this may take some time)
INFO SSH hostkey generated
INFO Environment set-up generated in /home/someuser/confd-X.X/
confdrc
INFO ConfD installation script finished
$
```

Many more installation related topics (e.g. uninstall) are covered in the installation root README file.

## Setting up Your Environment

In addition to unpacking the installation files, the installation program will create a file called `confdrc` with `PATH`, `MANPATH` and other handy environment settings for using this installation.

Sourcing this file makes it easier to use the installed ConfD:

```
$ source ~/confd-X.X/confdrc
$ man confd
...
```

If you are working with multiple versions of ConfD, just source the `confdrc` file in each installation to switch between them.

## Finding Documentation

At the root of the ConfD installation, there is a `doc/` directory containing the main ConfD User's Guide in `.pdf` and `.html` formats. These also contain the complete set of man pages for all the ConfD APIs. The man pages can also be read using the `man` command. The Java APIs are also documented in `javadoc` in there.

Also at the root of the ConfD installation there is a `README` file with some practical hints and pointers for hands-on users.

One of the best sources of usage information is the examples library found in the `examples.confd/` directory. It contains about 80 example applications demonstrating the use of various ConfD APIs. Each example has a `README` file explaining the purpose and usage of the example.

Apart from the man pages, extensive information about command line options can be obtained by running the `confd` and `confdc` commands with the `--help` flag.

## Launching a First Example

All examples have a certain basic structure in common. At the root of each example there is a `README` file explaining the purpose and how to build and run the example. All examples have a `Makefile` that supports targets `clean`, `all` and `start`.

The biggest example in the examples library is the Quagga example. Unlike the other examples, it is not focused on a single small use case or API. Instead it demonstrates what a small system consisting of multiple independent applications would look like

## Watching a Canned Demo

The Quagga demo is what we typically use to describe what ConfD is and does. The `README` file above gives you a good quick tour of what ConfD provides. The tour of the Quagga demo example can also be seen on-line. Navigate to [www.tail-f.com](http://www.tail-f.com), select the ConfD product section and click the Technical Demos tab.

### TO RUN IT:

1. Install ConfD as described above. Here we assume ConfD was installed in the directory `~/confd-X.X/`
2. Type `./confd-X.X/confdrc` – this sources the `confdrc` file to set up the `PATH` etc
3. Type `cd ~/confd-X.X/examples.confd/demo/quagga`
4. Type `make clean all` – this builds the Quagga demo applications
5. Type `make start` – this starts ConfD and the Quagga demo applications
6. Direct your web browser to the URL `https://localhost:8888`
7. In your web browser, log in as `admin / admin`
8. Then follow the guided tour by looking at the instructions in the `~/confd-X.X/examples.confd/demo/quagga/README` file

# ConfD Anatomy & Structure

ConfD is an on-device management framework. Its main purpose is to enable device manufacturers to quickly provide end user operators and higher-level management systems a set of rich, high-quality interfaces to manage the device.

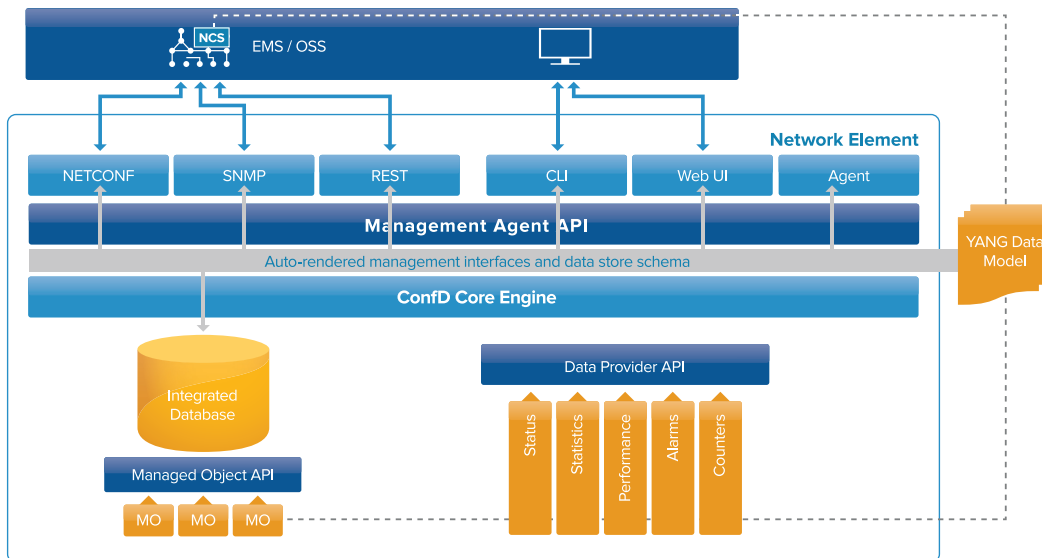


Figure 1: ConfD block diagram

At the top of the diagram, on top of ConfD, the block labeled EMS/OSS and the terminal icon represents higher-level management systems end user operators. They are the consumers of the management interfaces presented by the device through ConfD.

Moving inside the ConfD block, the top layer is the ConfD management interfaces. NETCONF, SNMP and REST are popular machine-to-machine device management protocols. Command Line Interface (CLI) and Web User Interface (Web UI) are well suited for human operators. The “Other” block represents the possibility to add other management interfaces beside the ones provided by some programming.

These management interfaces are often said to be “northbound”, since they are facing upwards on the picture. The device developer can choose which of the management interfaces to include and expose in his device.

In the middle there is the ConfD Core Engine. This block represents the heart and the majority of the code in the ConfD framework. Find more details around the functionality in “The Core Engine and CDB Database” section.

Looking “southbound”, i.e. at the bottom of the diagram, you find the application interfaces. MO stands for Managed Object, a very old standard term for the applications being managed.

## Management Interfaces

The ConfD framework is **Model Driven**. This means that ConfD automatically renders all the management interfaces from a data-model. A default rendering of each interface is produced automatically without any programming at all, solely based on the model. If this model is updated, that is automatically reflected in all management interfaces. The “YANG Modeling” section discusses this in some more depth.

Another key concept with ConfD is the support for **Transactions**. Most of the northbound interfaces provide transaction-safe configuration changes. This greatly simplifies the operator’s use of the device. The value and meaning of transactions is discussed in the “Brief Transaction Theory” section.

ConfD allows any number of management systems and operators to be logged-in and make configuration changes, show status, invoke management actions, etc. at the same time. They may use any mix of the management interfaces in parallel.

Here is a brief presentation of the available human user interfaces:

Traditionally, it has been common to have a separate “stove-pipe” (isolated silo) for each management interface, with different structure and functionality. This is not so with ConfD.

- **Cisco IOS style CLI:** This CLI style is similar to the CLI found on classic devices like Cisco 7200 and many others. This CLI is not transactional, which means each command takes effect immediately as the operator hits ENTER.
- **Cisco IOS XR style CLI:** This CLI style is similar to the CLI found on Cisco’s high-end equipment, like ASR9k and CRS-1. This CLI is transactional, which means the operator can make a lot of configuration changes, and then activate them with the commit command.
- **Juniper style CLI:** This CLI is similar to the one found on all Juniper’s equipment and many others. This CLI is transactional. It is often considered to be the CLI style easiest to use.
- **Web UI:** This is a rich client “Web 2.0” style user interface, directly rendered from the data-model. There is also a Web UI development framework to enrich and customize the auto-rendered user interface with additional functionality or altered behavior, if desired. The WebUI is rendered inside the browser client allowing for dynamic model-changes. Runs in an embedded Web server over http and/or https.

The CLI rendering engine is one of the strong points of ConfD. Using just the YANG data-model as input, ConfD renders CLI interfaces that are endorsed by network engineers. Using special YANG annotations in the models, the developer can give extra directions to the renderer on the desired behavior or certain command or mode. All the CLI variants support rich command editing with tab-completion, history, hints and help.

Now, let’s look at a brief presentation of the available machine-to-machine interfaces:

- **SNMP:** This is the most popular network management protocol for monitoring and fault handling. The ConfD SNMP agent supports SNMPv1, SNMPv2c, and SNMPv3. SNMP is not transactional, but ConfD treats each PDU as a transaction.
- **REST:** This is a popular design pattern for scripting towards a device. It provides a resource-based interface using the HTTP verbs and content in XML or JSON to modify resources. REST is not transactional, but ConfD treats each REST message as a transaction.

- **NETCONF:** This is the most powerful management protocol supported by ConfD and very popular in Software Defined Networking (SDN) contexts. ConfD implements the full NETCONF specification including all optional parts, including support for network-wide transactions, multiple data stores and XPATH queries. It runs over SSH with content encoded in XML.

## YANG Modeling

The first thing a developer would do when starting to develop the management interfaces for devices using ConfD would be to create a YANG Data Model.

As soon as a YANG model with the above information has been created, it can be compiled and loaded into ConfD. ConfD will then automatically render all of the described elements in all the management interfaces.

Say your device has a set of interfaces. Each interface has a number of properties that can be configured and monitored, as well as actions that can be executed.

All of these things would be modeled in YANG. The model might say there is a list of interfaces. The interfaces are identified by name. The name is an alphanumeric string of at most 20 characters, not starting with a number. Each interface has a configurable desired port speed, selectable from some set of possible values, one of which is “auto”.

An interface also has several status a.k.a. operational properties. These properties can be observed, but not directly set by the operator. One is the actual port speed; a couple of others include byte counters for ingress and egress traffic.

There would also be an action to reset all the interface byte counters to zero.

An actual YANG model according to the above could look like this:

```
list interface {
  key name;
  leaf name {
    type string {
      length 1..20;
      pattern "[a-zA-Z][0-9a-zA-Z_:/-]+";
    }
  }
  leaf port-speed {
    type enumeration {
      enum 100M;
      enum 1G;
      enum 10G;
      enum auto;
    }
  }
  container status {
    config false;
    leaf actual-speed {
      type uint64;
      units "bits/s";
    }
    leaf ingress-bytes {
```

### The YANG Data Model describes:

- Everything that can be configured on the device
- Everything that can be monitored on the device
- All the administrative actions that can be executed on the device
- All the notifications the device may generate

```

        type uint64;
    }
    leaf egress-bytes {
        type uint64;
    }
}
rpc reset-interface-counters;

```

Figure 2: Example YANG model

Based on a YANG model like this, ConfD would render a Cisco IOS and IOS XR style CLI with commands to set, no & show the configuration properties, show the operational properties and a command to execute the action. In the Juniper CLI there would be commands to set, delete, edit and show all the objects, etc.

ConfD would also render a Web UI with table view of the interfaces, entry fields or drop downs for the configuration properties, current values for the operational properties and a push button to execute the action. ConfD would also render an SNMP interface (including a MIB), a REST interface and a NETCONF interface to all this. No programming would be required to get any of this.

The YANG modeling language is a standard defined by IETF in the document RFC 6020. IETF is the same organization that defined SNMP, SMI and the standard MIBs many years ago.

One key aspect of YANG is the clear **separation of configuration and operational data**. Separating them makes it easy for an operator to retrieve only the configuration data of a device, and restore it at a later date, or modify it slightly and apply it to another similar device.

SNMP is not designed this way, which is one of several reasons SNMP is rarely used for configuration.

YANG modeling is an art, and models can be designed in many ways. It is possible to construct YANG models that mimic an existing behavior, e.g. a legacy CLI, very closely. In general, however, a more versatile and long lasting device management interface results if the YANG model is designed in a more generic fashion.

## Brief Transaction Theory

A transaction is a set of actions that fulfills the following four properties, collectively referred to as the ACID properties:

- **Atomicity:** A transaction is always carried out completely or not at all. This principle is widely known.
- **Consistency:** Transactions are instantaneous, i.e. the actions within a transaction are all logically happening at the same time. Transaction T1 containing actions A and B is identical to transaction T2 containing actions B and A. The transaction manager must treat T1 and T2 exactly the same, or else it does not fulfill the ACID properties, i.e. the system does not support transactions.

The YANG modeling language is a standard defined by IETF in the document RFC 6020. IETF is the same organization that defined SNMP, SMI and the standard MIBs many years ago.



- **Independence:** All transactions executed by a transaction manager must be executed in sequence, one at a time. Or so it must appear to all observers.
- **Durability:** Once a transaction has been completed, it must stick, regardless of power failures or other incidents in scope of the transaction manager specification.

IETF investigated what the operator requirements are on Network Management back in 2001-2002, and wrote a nice informational RFC document about their findings. It is still very relevant today. With its less than 20 pages, high-level, well written outlook, it's an excellent summary: <http://tools.ietf.org/html/rfc3535>

In this document, operators say that transactions and **network-wide transactions** are a key mechanism to automate the network management, i.e. transactions within a single device, and even better that cover more than one device.

Without transactions, operators and Network Management Systems (NMS) would have to “know” in which sequence all configuration changes should be sent to the devices in the network. With the introduction of transactions, this sequencing information no longer comes from the operators or NMS systems. A transaction is all-at-once, so there is no internal sequence in a transaction. Applications inside the device, however, depend heavily on receiving configuration change events in a particular order. So, in ConfD, **applications specify in which order they would like to receive configuration change notifications**, regardless of how the operator or NMS gave it to the system.

An application always wants to know about any interface creations or modifications first, then any routing changes over those interfaces, and finally any interface deletions would specify that in its database subscription calls. ConfD would then deliver this information in that sequence, regardless of how it was given by the NMS or operator.

Without transactions, operators and NMS systems would also have to undo everything they did so far if some big change failed half way through. This is complicated, expensive, error prone and very difficult to test. With transactions, this **error handling code is not needed** in the NMS, leading to great savings.

For example, when an NMS was setting up an L3VPN, say, for some client across a network of routers, firewalls and deep packet inspection devices from six vendors, if something went wrong, the application would have to “know” how to abort the change and undo all that was done to the failing device. It would also have to revert all the changes already successfully made to the devices prior to the failure.

With the introduction of transactions, all these burdens on the NMS go away. In a typical, mature NMS application, that would be more than 50% of the code. No wonder operators are requiring that network equipment support network wide transactions.

A somewhat similar situation was noted in the database industry in the 1980's. Then the transaction concept was widely adopted there, and the industry transformed permanently.

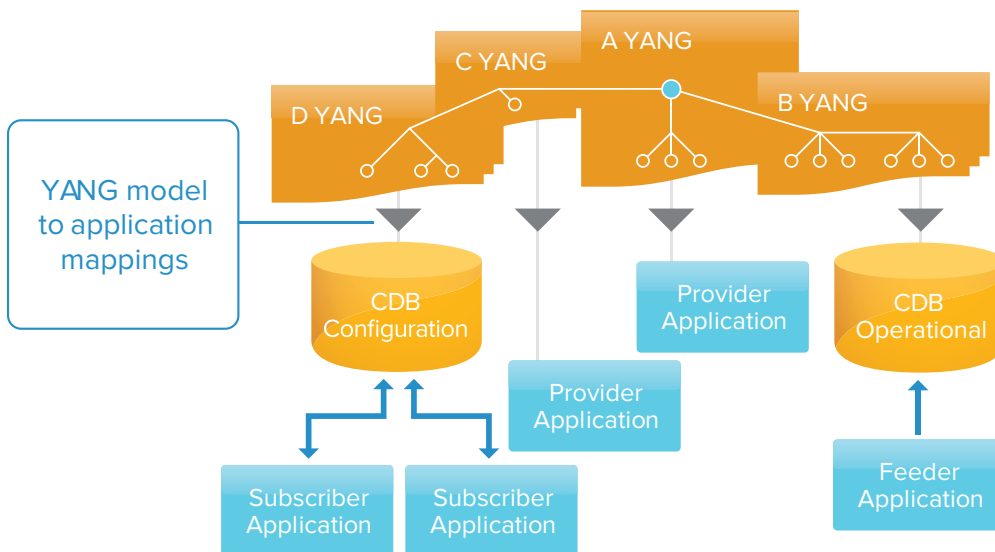
In this document, operators say that transactions and **network-wide transactions** are a key mechanism to automate the network management, i.e. transactions within a single device, and even better that cover more than one device.

## The Core Engine and CDB Database

The ConfD Core Engine is what we call the collection of all the main subsystems in ConfD. Around 80% of the ConfD code, functionality and features are here. The management protocol agents are relatively thin layers on top of the ConfD Core Engine.

The main subsystems are the Transaction manager, the Data Model manager, the Access Control manager, the Logging subsystem, The High Availability subsystem, the Rollback generator, and the CDB Database.

The Data Model manager keeps track of all the YANG data models, and the tree of managed objects they jointly describe. For each part of the model, it keeps track of which application or system component is the responsible data provider for this part of the model. This is similar to how UNIX file systems are mounted within each other and responsible for one or several sub-trees.



**Figure 3: YANG models describing both configuration and status information are mounted at the root and on top of each other. Every list and leaf in the models are mapped to a provider, either a database such as CDB, or provider applications. Subscriber applications are notified of changes to relevant parts of the configuration. Feeder applications inject operational data, e.g. performance data into the CDB operational data store.**

For configuration data, some sort of database would be the data provider. In most cases, the built-in CDB Database is used, but other databases could be plugged-in, or even some simple storage in flat files could be built.

For operational data, i.e. status information, the applications themselves register as data providers most of the time. There is usually no point in storing this kind of potentially rapidly changing data in a database. The values are simply computed when an operator wants to see them. For certain types of operational data, e.g. performance and alarm data, it may make sense to store this in a database. The CDB Database has a separate Operational Datastore available for this purpose, should you wish to use it.

The Data Model manager uses special YANG model annotations to know which application or system component is responsible for each part of the data model.

The CDB Database is a transactional, ACID compliant database where the schema is composed of the YANG models. CDB picks up any changes to the model automatically and translates any existing contents in the database to the new schema.

CDB is a no-SQL hierarchical database. This makes it easy to represent the configuration data for the device, which is hierarchical in nature. This avoids the object-relational-mapping problem from hierarchical data to say SQL tables. CDB is built around ACID transactions, so any configuration change either happens completely or nothing is changed in the data store.

The key features of CDB are:

- **Persistent embedded in-memory database** with optimized performance for hierarchical data. The in-memory model makes all validation, CLI tab-completion etc. execute quickly. The CDB journals are stored to disk.
- **Automatic schema management:** the embedded data store removes all traditional work around databases. CDB uses YANG as native schema language, so the schema management is transparent. Nor is there any need for run-time configuration or administration of the database.
- **Schema upgrade/downgrade:** the schema (YANG model) can be changed in run-time. CDB will automatically manage migrating the existing stored data. In case of conflicting schema and instance data, upgrade hooks can be registered.
- **Transactions:** the data store and the ConfD transaction engine are tightly connected. This removes the burden from application developers to understand and implement transactional integrity.
- **Subscription-based programming model:** programmers can subscribe to changes to CDB and react upon those. This in contrast to stub-based approaches. It also enables easy addition of logic.
- **High-availability:** CDB can run in high-availability mode, where hot-standby nodes are replicated from the master node.
- Support for **candidate** and **running data** stores
- **Fast and light-weight:** CDB is optimized for embedded applications, the foot-print is small and response times are quick.

The ConfD Core Engine also manages AAA, i.e. Authentication, Authorization and Accounting/Auditing. The list of users and their passwords or cryptographic tokens can be kept in the CDB database, in the underlying operating system, e.g. PAM, in remote Radius/Tacacs+/LDAP etc servers or a combination of this. Fine- or coarse-grained access control rules can be defined per group/role and users can be assigned to roles/groups.

Logging can be set up to create detailed audit logs, or system level or application tracing. The information may be logged to local files, remote syslog or custom applications.

Logging can be set up to create detailed audit logs, or system level or application tracing. The information may be logged to local files, remote syslog or custom applications.

## Application APIs

Applications a.k.a. managed objects running south of ConfD typically don't know or care which management interface a particular operator request is coming from. All they see is a request from ConfD to get some value, or a transaction with changes, which should be validated or committed.

The model-driven nature of ConfD is illustrated below. You write a data-model for your device, load that to ConfD and you have an agent with all northbound API, persistent configuration data store and all other features of the core engine.

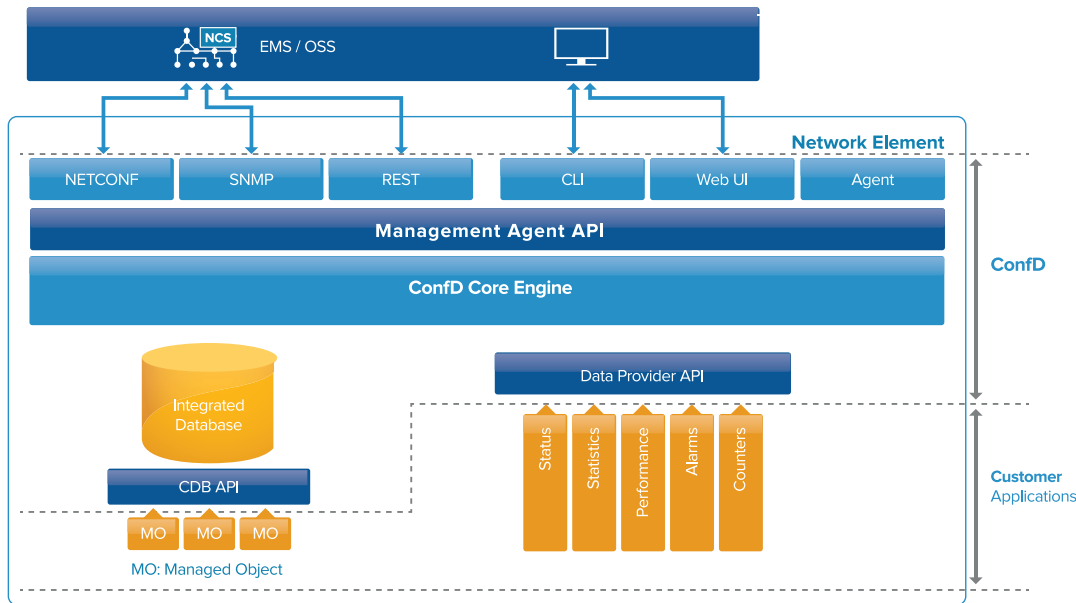


Figure 4: The ConfD Core Engine in the middle, with the Management Agent API (MAAPI) on top and CDBAPI and Data Provider API (DPAPI) below.

There are three main APIs in ConfD:

- **MAAPI:** Management Agent API. With this API applications can start transactions, read and write configuration data, read operational status data or invoke actions the same way as an operator could. In fact, this is the API that the management agents use in their implementation.
- **DP API:** Data Provider API. With this API applications register for callbacks for various kinds of events, e.g. when an operator is asking to see the current value of some operational data value, e.g. the CPU temperature.
- **CDB API:** Database API. With this API applications read configuration data from the database and subscribe to configuration changes. Applications may also write operational data into the operational data store in CDB.

There is a wealth of different kinds of application that might be running inside a ConfD based system. How they are divided into different executables and potentially spread out over a set of nodes in a cluster is up to the software designer. They could all run in a single executable, or run in a large number of separate processes across a wide area network. Here's a menu of the most common application types:

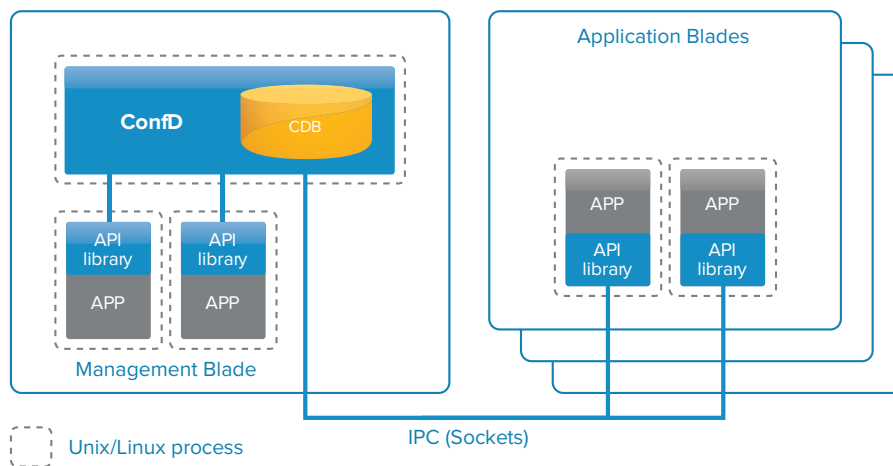
- **Subscriber:** An application using the CDBAPI that has requested to be notified when an operator modifies parts of the configuration that the application is interested in, e.g. set hostname or add an entry to the interface table.
- **Data Provider:** An application that knows the current value of something in the data model. ConfD will contact this application through the DPAPI to get the current value when requested by an operator, e.g. show CPU temperature.
- **Action Provider:** An application that implements an action listed in the data model. ConfD will contact this application through the DPAPI when the operator executes the action, e.g. reset interface byte counters.
- **Validator:** An application that participates in the validation, i.e. determining if the configuration that results when applying a certain transaction is valid. A validator can respond “Yes, this is fine”, “No, this configuration is invalid because ...” or “Warning: This configuration is valid, but are you aware that ... (proceed/abort) ?” Validators use the MAAPI since they need to see the upcoming configuration the same way an operator sees it, not the configuration currently stored in the database.
- **Transaction hook and set hook:** An application that is invoked when an operator wants to commit a transaction, before the validation starts (transaction hook), or each time the operator sets a value in a transaction (set hook). The application can then modify or extend the transaction with further modifications on behalf of the operator. These modifications are usually done in parts of the data model that are invisible to the operator. Hooks use the MAAPI since they are modifying transactions on behalf of the operator.
- **Data feeder:** An application that is repeatedly storing operational data, e.g. performance readings, into a database table using the CDBAPI.
- **Two-phase subscriber:** A subscriber application that is also participating in the validation phase using the CDBAPI. It can thereby reject upcoming configuration changes.
- **Transform:** An application that translates an externally visible data model into an internal data model. This is useful for example when an application that uses an SNMP MIB like API internally needs to be dressed up in a user-friendlier Cisco-like CLI command tree. Transforms use both the DPAPI and MAAPI.
- **Authentication and authorization callback:** Application that participates in the authentication or authorization of operators in the system. They use the system APIs, not listed as one of the main APIs above.
- **Upgrade client:** An application that participates in a software upgrade campaign to update the data stored in CDB beyond the automatic conversion.
- **Policy, Command and Post-commit script:** Scripts to enforce policies or perform additional actions at commit time or when specific commands are executed.
- **Completion plug in:** An application that alters the standard TAB-completion behavior in the Command Line Interfaces.
- **User Type plug in:** An application that implements an additional base type that can be used in the YANG models.

- **Message formatters:** An application that reformats or translates operator messages, e.g. error messages.
- **Inter-process communications plug in:** An application to replace the default IPC mechanism (TCP sockets) used between ConfD and clients. Useful if clients need to use for example UNIX-domain sockets or TIPC to reach ConfD.
- **Alarm manager:** An application that receives events from many applications in the system, and decides which alarm states should be raised and cleared in the system, and how those alarm states should be communicated to operators and management systems. E.g. do nothing, send an SNMP notification, SNMP notification plus NETCONF notification plus message to all operators, etc.

Beyond this list, there are often customizations and extensions to the way the CLI and WebUI behave.

Many management agent frameworks are stub-based. This means that a command list or data-model is fed to a compiler, which generates stub functions that must be implemented, “instrumentation”. The instrumentation code must handle all application logic like persistence, error-handling, mapping to the underlying resources etc.

ConfD turns this upside-down. ConfD is a complete agent and uses a **subscription-based** programming style, based on the YANG data-model. Northbound APIs and User Interfaces are run-time rendered. Configuration changes are made persistent into the built-in data store. As a developer you can iteratively add subscriptions to the configuration changes and map this to the underlying resources. This has several benefits including iterative/agile development and core features taken care of by the ConfD Core Engine. The developer can focus on the application logic.



The most common run-time deployment of ConfD is a scenario distributed over a management blade and several application blades:

Figure 5: Management and Application Blades

The application blades link with the ConfD API library `libconfd` and the communication is handled over sockets to the ConfD agent.

The APIs come with different language bindings:

- C
- Java
- Erlang
- Python (Currently MAAPI only)

## High-Availability

ConfD supports replication of the CDB configuration and some or all CDB operational data. The replication architecture is that of one active master and a number of slaves. All configuration write operations must occur at the master and ConfD will automatically distribute the configuration updates to the set of live slaves, synchronously or asynchronously. Operational data in CDB may be replicated or not based on the `tailf:persistent` statement in the data model and the ConfD configuration. All write operations for replicated operational data must occur at the master, with the updates distributed to the live slaves, whereas non-replicated operational data can also be written on the slaves.

ConfD does not take any decisions regarding which node is slave or master. It relies on an external framework calling functions:

```
be_master()
```

```
be_slave()
```

## SNMP Considerations

Even though SNMP is not used very often for configuration and provisioning purposes, it is the dominating protocol for network monitoring and fault management. For many device types, an SNMP agent implementation is a hard requirement.

ConfD has a built-in SNMP Agent with full support for any combination of SNMPv1, SNMPv2c, and SNMPv3 including the framework MIBs, authentication and encryption.

In the same manner as for all other northbound APIs, SNMP is not a separate stove-pipe/silo. It is a view of the data-model managed by ConfD. ConfD supports tools that can render YANG modules from MIBs and vice versa.

There might be different starting points when it comes to SNMP:

1. There are standard or proprietary MIBs that must be supported
2. There are YANG modules that must be supported, we just need a MIB representation for this information
3. Or a mix of above

### Scenario 1: MIBs are given

The YANG modules and associations are generated with the `confdc --mib2yang` translator program. The generated YANG modules will in this case resemble the original MIBs in terms of structure, naming and level of details.

You may want to take a look at the `examples.confcd/snmpa/2-mib-to-yang` for a deeper dive.

### Scenario 2: YANG modules are given

A YANG module may be translated into a MIB using the `confdc --emit-mib` command. The generated MIB will use the same names as the YANG module, but the deep structure with tables inside tables common in YANG will be translated to flat tables on the top level (as required in SNMP) with foreign keys etc. added. The YANG module may be annotated with SNMP names and OID values, in which case those names and OIDs will be used in SNMP.

### Scenario 3: YANG and MIBs are given

In this case, MIB associations should be written to bind MIB objects to the nodes in the YANG data model. YANG annotations statements `tailf:snmp-name`, `tailf:snmp-oid`, etc. are added either directly in the YANG module or in a separate annotation file. In many cases the mapping can be more complex than can be expressed by annotations. In such cases a transform application is required in order to implement the translation.

# Running ConfD

Let's walk through one of the examples in the ConfD examples collection.

You will find this example in `examples.confD/intro/1-2-3-start-query-model`. This example implements the simplest possible subscriber. At startup it reads out the dhcp-related configuration and writes the Linux dhcpd config file. Then it subscribes to configuration changes in the dhcp-area of the YANG model, and whenever an operator makes any changes here, ConfD notifies the subscriber. The subscriber reads all the dhcp related configuration and regenerates the entire dhcpd config file again. A very basic approach to configuration subscriptions.

## Data-Model Definition

The first step with ConfD is always to write a data model that specifies the configuration and operational data. Let's look at the DHCP YANG data-model step-by-step.

All YANG modules define module related meta-data:

```
module dhcpd {
  namespace "http://tail-f.com/ns/example/dhcpd";
  prefix dhcpd;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-xsd-types {
    prefix xs;
  }

  typedef loglevel {
    type enumeration {
      enum kern;
      enum mail;
      enum local7;
    }
  }
}
```

Figure 6: DHCP YANG Module, snippet #1

This snippet gives a name to the module along with a unique namespace. Useful data types from IETF and Tail-f are imported and finally a typedef for loglevels is defined.

Next we have the core part of the data-model:

```
container dhcp {
  leaf default-lease-time {
    type xs:duration;
    default PT600S;
  }
  leaf max-lease-time {
    type xs:duration;
    default PT7200S;
  }
}
```



```

leaf log-facility {
    type loglevel;
    default local7;
}
container subnets {
    uses subnet;
}
container shared-networks {
    list shared-network {
        key name;
        max-elements 1024;
        leaf name {
            type string;
        }
        container subnets {
            uses subnet;
        }
    }
}
}
}

```

Figure 7: DHCP YANG Module, snippet #2

The data-model defines configurable attributes for default lease time, max lease time and log facility. The subNets container uses the subNet definition. You can think of this as a macro expansion. The definition will be shown next. Finally the data-model defines a list of shared-networks. Again, note the uses construct of the subNet grouping.

Finally we can study the definition of the “subNet” grouping:

```

grouping subnet {
    list subnet {
        key "net mask";
        leaf net {
            type inet:ipv4-address;
        }
        leaf mask {
            type inet:ipv4-address;
        }
        container range {
            presence "Enable specific range for subnet";
            leaf dynamic-bootp {
                type boolean;
                default false;
                description "Enable BOOTP for this subnet";
            }
            leaf low-addr {
                type inet:ipv4-address;
                mandatory true;
                description "Lowest address in range";
            }
            leaf hi-addr {
                type inet:ipv4-address;
            }
        }
    }
}

```

```

        description "Highest address in range";
    }
}

leaf routers {
    type string;
}

leaf max-lease-time {
    type xs:duration;
    default PT7200S;
}
}
}

```

Figure 8: DHCP YANG Module Snippet #3

It defines a list of subnets, keyed with address and mask. Each subnet BOOTP can be configured as true or false.

Since YANG is an open standard with a canonical textual representation (defined in RFC6020) it can be authored in any editor like Vi, Emacs or Eclipse, etc. guided with modes or plug-ins.

There are also visual editors available. An example from MG-SOFT is given below:

Since YANG is an open standard with a canonical textual representation (defined in RFC6020) it can be authored in any editor like Vi, Emacs or Eclipse, etc. guided with modes or plug-ins.

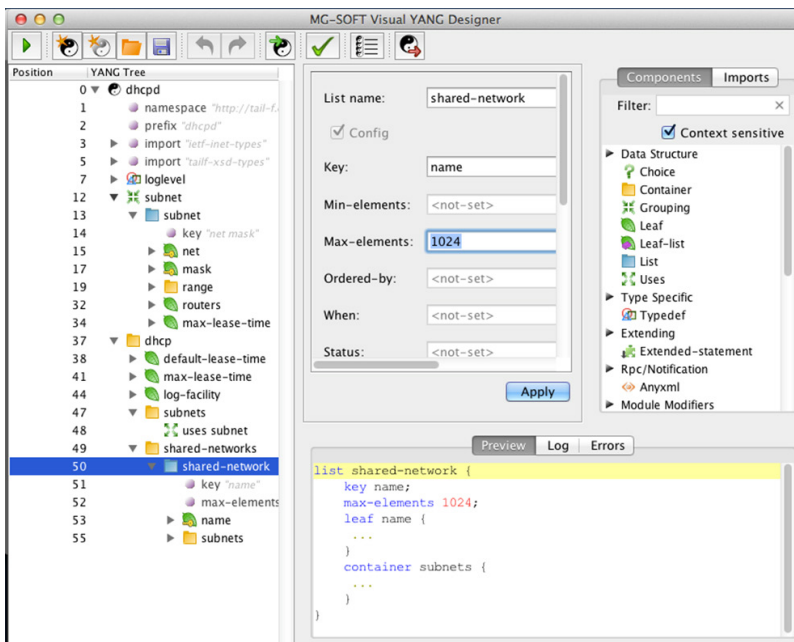


Figure 9: YANG Visual Editor

The good thing about textual modeling languages is that it leaves room for any tools to be built. A useful open source tool is pyang<sup>1</sup>, which can do lots of things based on YANG input. One of the most useful is rendering a tree structure:

```
module: dhcpd
  +--rw dhcp
    +--rw default-lease-time?   xs:duration
    +--rw max-lease-time?       xs:duration
    +--rw log-facility?         loglevel
    +--rw subnets
      | +--rw subnet* [net mask]
      |   +--rw net             inet:ipv4-address
      |   +--rw mask             inet:ipv4-address
      |   +--rw range!
      |     | +--rw dynamic-bootp?  boolean
      |     | +--rw low-addr         inet:ipv4-address
      |     | +--rw hi-addr?        inet:ipv4-address
      |     +--rw routers?         string
      |     +--rw max-lease-time?   xs:duration
    +--rw shared-networks
      +--rw shared-network* [name]
      +--rw name               string
      +--rw subnets
        +--rw subnet* [net mask]
          +--rw net             inet:ipv4-address
          +--rw mask             inet:ipv4-address
          +--rw range!
            | +--rw dynamic-bootp?  boolean
            | +--rw low-addr         inet:ipv4-address
            | +--rw hi-addr?        inet:ipv4-address
            +--rw routers?         string
            +--rw max-lease-time?   xs:duration
```

Figure 10: pyang output

<sup>1</sup> SNMP Geeks: this is similar to smilint/smidump.

## Data-Model Compilation

The `dhcpd.yang` module can now be compiled to the ConfD load format `dhcpd.fxs`:

```
$ confdc -c dhcpd.yang
```

ConfD reads a file called `confd.conf` when it starts. One of the settings is `load-path` – where to look for `.fxs` files. All that is needed now is that the generated `dhcpd.fxs` file is in the load path when starting ConfD.

So you can start ConfD:

```
$ confd
```

## Rendered Management Interfaces

At this point we have all the management interfaces and persistent storage of configuration data up and running directly generated from the data-model. The Web UI is shown below:

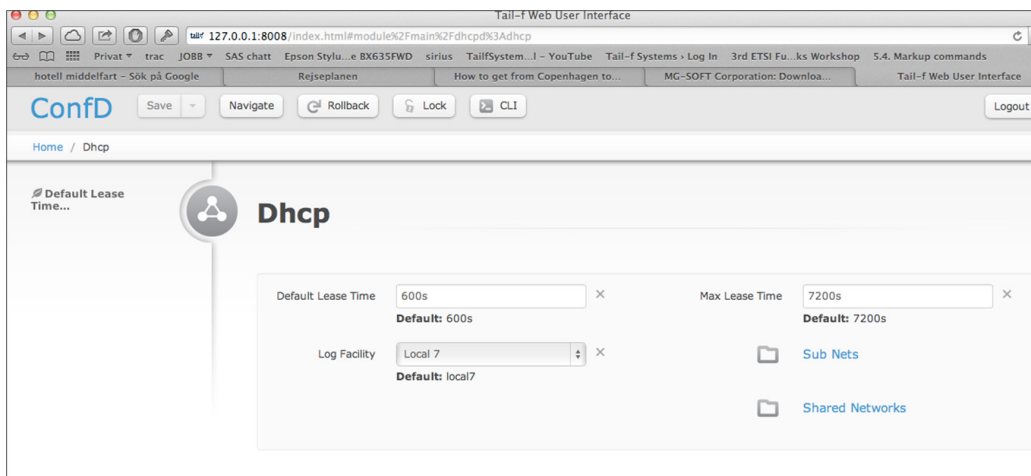


Figure 11: Auto-rendered DHCP Web UI

Starting the CLI in IOS mode (-I for IOS-style, -C for IOS-XR style, -J for Junos-style):

```
$ confd_cli -I -u admin
admin connected from 127.0.0.1 using console on wallair.local
confd> enable
confd# configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
confd(config)# dhcp
Possible completions:
  default-lease-time  log-facility  max-lease-time
  shared-networks    subnets
```

Figure 12: ConfD IOS CLI

Or REST for that matter:

```
$ curl http://localhost:8008/api/running/dhcp --header "Accept:
application/vnd.yang.data+json" -u admin:admin
{
  "dhcp": {
    "_self": "/api/running/dhcp",
    "_path": "/dhcpd:dhcp",
    "default-lease-time": "PT1H",
    "subnets": {
      "_self": "/api/running/dhcp/subnets",
      "subnet": [
        {
          "_self": "/api/running/dhcp/subnets/subnet/
192.168.128.0%2C255.255.255.0",
          "net": "192.168.128.0",
          "mask": "255.255.255.0"
        }
      ]
    },
    "shared-networks": {
      "_self": "/api/running/dhcp/shared-networks",
      "shared-network": [
      ]
    }
  }
}
```

Figure 13: ConfD REST Example

## Configuration Data-Store, Rollbacks

We will now illustrate that all configuration changes are persistent and available as roll-backs. If we first use the Web UI to change default lease and max lease time:

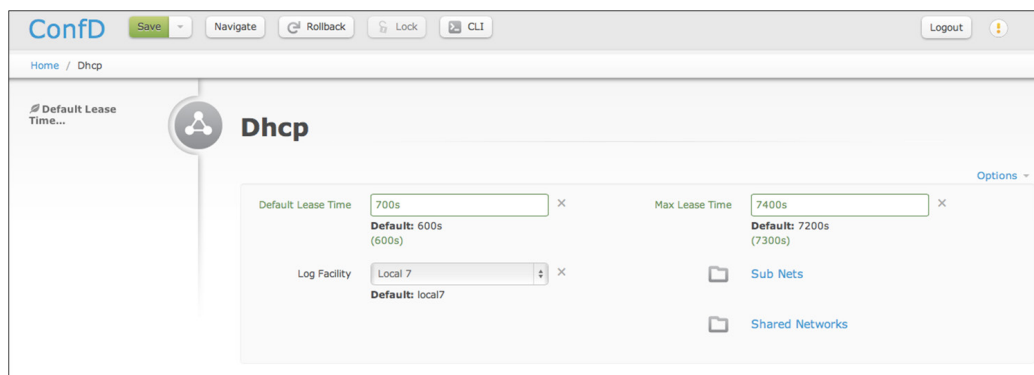


Figure 14: Web UI configuring DHCP

Pressing “Save” implies a transactional commit. The rollback button gives access to all changes to the system:

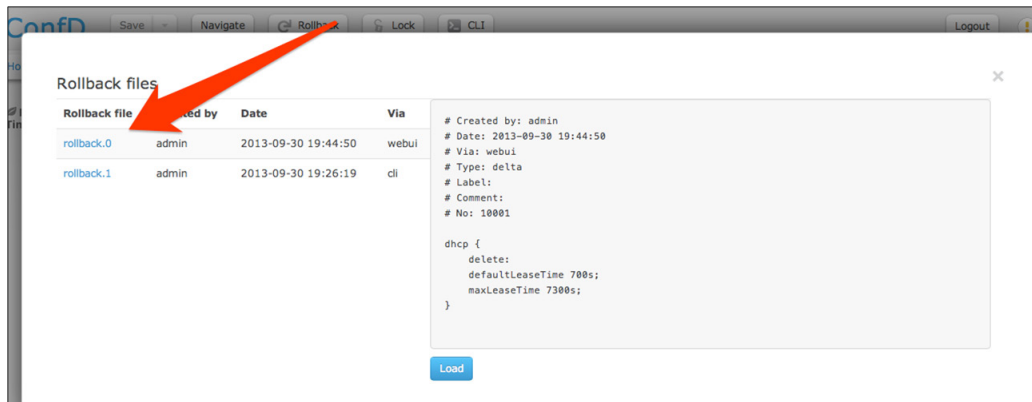
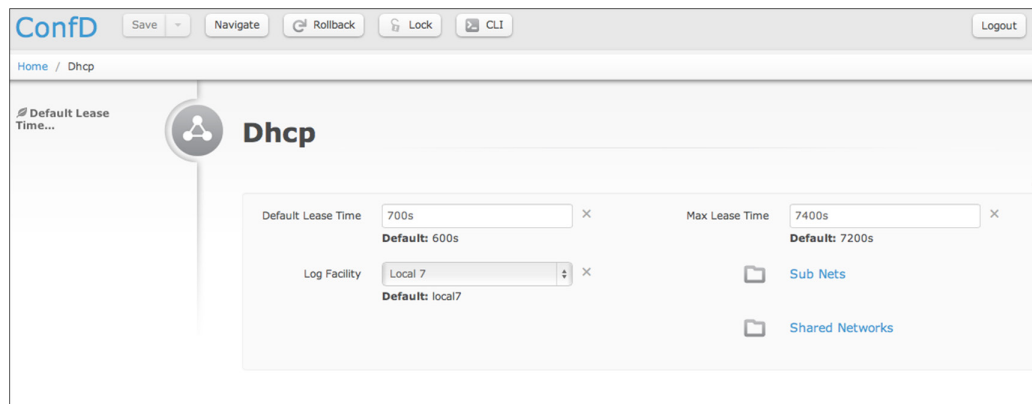


Figure 15: Selecting a configuration rollback

If we change it again to non-default values:



Save the configuration. In order to demonstrate that these settings survive a restart of ConfD, try stopping ConfD, and then start ConfD again and inspect the values over CLI for example. Below we use the Juniper-style CLI:

```

confd> configure
Entering configuration mode private
[ok][2013-12-07 19:35:45]

[edit]
confd% show dhcp
default-lease-time 1h;
subnets {
  subnet 192.168.128.0 255.255.255.0 {
    range {
      low-addr 192.168.128.60;
      hi-addr 192.168.128.89;
    }
  }
}
[ok][2013-12-07 19:35:50]
    
```

Figure 17: Showing the change in the Juniper style CLI

## Making the Configurations Changes Happen

So far we have shown that ConfD renders various management interfaces and provides persistent storage of the configuration data. A brief introduction to the core features was also given. The most important remaining question at this point is: ***How do I connect this to my application?***

Going back to our DHCP example, an operator might configure the DHCP subsystem as follows:

```
default-lease-time 600s;
max-lease-time 7200s;
subnet 192.168.128.0 netmask 255.255.255.0 {
    range 192.168.128.60 192.168.128.98;
}
shared-network 22429 {
    subnet 10.17.224.0 netmask 255.255.255.0 {
        routers rtr224.example.org;
    }
    subnet 10.0.29.0 netmask 255.255.255.0 {
        routers rtr29.example.org;
    }
}
```

Figure 18: DHCP Configuration File

An application inside the device would then subscribe to this part of the YANG model, and whenever there were any changes to the DHCP part of the configuration in ConfD, regenerate a new `dhcpd.conf` and then restart the Linux DHCP server to make the new configuration take effect. That is the only way to make the standard Linux DHCP daemon read a new configuration.

```
int main(int argc, char **argv)
{
    struct sockaddr_in addr;
    int subsock;
    int status;
    int spoint;

    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    addr.sin_family = AF_INET;
    addr.sin_port = htons(CONFD_PORT);

    confd_init(argv[0], stderr, CONFD_TRACE);

    /*
     * Setup subscriptions
     */
    if ((subsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open socket\n");

    if (cdb_connect(subsock, CDB_SUBSCRIPTION_SOCKET,
                   (struct sockaddr*)&addr,
                   sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to cdb_connect() to confd \n");
```

```

if ((status = cdb_subscribe(subsock, 3, dhcpd__ns,
                          &spoint, "/dhcp")) != CONFD_OK) {
    fprintf(stderr, "Terminate: subscribe %d\n", status);
    exit(0);
}
if (cdb_subscribe_done(subsock) != CONFD_OK)
    confd_fatal("cdb_subscribe_done() failed");
printf("Subscription point = %d\n", spoint);

/*
 * Read initial config
 */
if ((status = read_conf(&addr)) != CONFD_OK) {
    fprintf(stderr, "Terminate: read_conf %d\n", status);
    exit(0);
}
rename("dhcpd.conf.tmp", "dhcpd.conf");
/* This is the place to HUP the daemon */

```

Figure 19: Subscribing to CDB Configuration Changes

The C-code above shows how to establish a socket to ConfD and subscribe to changes under a specific path (/dhcp) in CDB. `cdb_subscribe` is a call to a ConfD library function, `read_conf` is an application specific function shown below.

Whenever there is a change under the /dhcp path in the model we need to read the current configuration kept in CDB (snippet shown):

```

static int read_conf(struct sockaddr_in *addr)
{
    FILE *fp;
    struct confd_duration dur;
    int i, n, tmp;
    int rsock;

    if ((rsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open socket\n");

    if (cdb_connect(rsock, CDB_READ_SOCKET,
                  (struct sockaddr*)addr,
                  sizeof (struct sockaddr_in)) < 0)
        return CONFD_ERR;
    if (cdb_start_session(rsock, CDB_RUNNING) != CONFD_OK)
        return CONFD_ERR;
    cdb_set_namespace(rsock, dhcpd__ns);

    if ((fp = fopen("dhcpd.conf.tmp", "w")) == NULL) {
        cdb_close(rsock);
        return CONFD_ERR;
    }
    cdb_get_duration(rsock, &dur, "/dhcp/default-lease-time");

```



```

fprintf(fp, "default-lease-time %d\n",
        duration_to_secs(&dur));

cdb_get_duration(rsock, &dur, "/dhcp/max-lease-time");
fprintf(fp, "max-lease-time %d\n",
        duration_to_secs(&dur));

cdb_get_enum_value(rsock, &tmp, "/dhcp/log-facility");
switch (tmp) {
case dhcpd_kern:
    fprintf(fp, "log-facility kern\n");
    break;
case dhcpd_mail:
    fprintf(fp, "log-facility mail\n");
    break;
case dhcpd_local7:

```

Figure 20: Reading CDB Configuration

Note how the paths in the YANG model (see “Data-Model Definition”, page 16) are referenced above.

So, the instrumentation for configuration data is implemented by subscribing to specific paths in the data-model and makes the change happen. This loosely coupled programming model makes it very easy to work with configuration data. Note how ConfD seamlessly handles tricky things like:

- Persistence
- Transactions
- Roll-backs

No user code is needed for that.

This was a simple example that reads all relevant configuration every time anything changes. Usually, you would implement a more fine-grained strategy where ConfD delivers change information in the sequence required by each application. Have a look at [examples.conf/cdb\\_subscription/iter](#) and [examples.conf/cdb\\_subscription/twophase](#) for details.

## Operational Data Providers

There are two primary kinds of operational data:

- Data that is calculated when requested. ConfD uses callpoints to handle this.
- Data that is calculated at regular intervals and stored, e.g. in CDR

### Operational Data with Callpoints

Most operational data is typically not kept in a database, but read at runtime by instrumentation functions whenever the operator or management system wants to see it. This could be for example rapidly changing counters contained inside the managed objects themselves.

Instrumentation for configuration data is implemented by subscribing to specific paths in the data-model and makes the change happen. This loosely coupled programming model makes it very easy to work with configuration data.

The ConfD example `examples.conf/5-c_stats` demonstrates how to do this. It allows the operator to see the current contents of the ARP cache of the system. The YANG data model looks like this:

```
module arpe {
  namespace "http://tail-f.com/ns/example/arpe";
  prefix arpe;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-common {
    prefix tailf;
  }

  container arpentries {
    config false;
    tailf:callpoint arpe;
    list arpe {
      key "ip ifname";
      max-elements 1024;
      leaf ip {
        type inet:ip-address;
      }
      leaf ifname {
        type string;
      }
      leaf hwaddr {
        type string;
        mandatory true;
      }
      leaf permanent {
        type boolean;
        mandatory true;
      }
      leaf published {
        type boolean;
        mandatory true;
      }
    }
  }
}
```

Figure 21: YANG Data-Model for ARP Operational Data

The container `arpentries` has two important sub-statements:

- **config false:** YANG statement saying this is not configuration data, i.e. from the operator's point of view, it's read-only data maintained by the agent
- **tail-f:callpoint arpe:** Tail-f YANG extension stating a name that the instrumentation C-code will later register as provider for

The `arpe` callpoint will invoke callbacks in an external program that have registered as data provider for "arpe". There can be several different applications on the same device that register themselves as providers for different callpoints, and one application can register multiple callpoints.

The fundamental callback functions in the Data provider API for a callback are `get_next` and `get_elem`:

- **get\_next:** This callback is invoked repeatedly to find out which keys exist for a certain list. ConfD will invoke the callback as a means to iterate through all entries of the list, in this case all “arpe” entries.
- **get\_elem:** This callback is invoked by ConfD when ConfD needs to read the actual value of a leaf element.

There are a dozen further callbacks that may make sense to implement. Many callbacks are optional, but may be provided as an optimization, e.g. to allow ConfD to ask for an entire list row at once, or quickly find a certain position in a list.

We need to step back here to introduce sessions and transactions. A user *session* corresponds directly to an SSH/SSL session from a management station to ConfD. A user session is associated with the IP address of the management station and the user name of the user who started the session, independently of which management agent is involved.

The user session data is always available to all callback functions.

A new *transaction* is started whenever an agent tries to access some data, e.g. read operational data. For each transaction two user-defined callbacks are potentially invoked:

- **init:** this callback will be invoked when a transaction starts.
- **finish:** This callback gets invoked at the end of the transaction. This is a good place to deallocate any local resources for the transaction.

We can now put the puzzle together. The tasks for the main function in our application are:

1. Initialize the ConfD library
2. Create a control and worker socket: whenever a new session is created (a user logs in), a request to initialize the sessions arrives from ConfD on the control socket. All further requests and replies for this session will be sent on the worker socket created or pointed to by the session `init` function.
3. Register the `init`, `finish`, `get_next` and `get_elem` callbacks

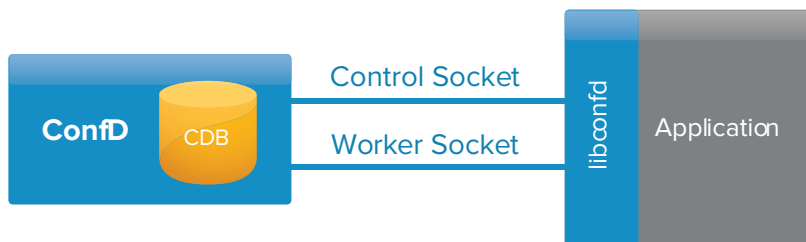


Figure 22: The control and worker sockets

When a user session is started, the `init` function is called (lazily, i.e. not until needed):

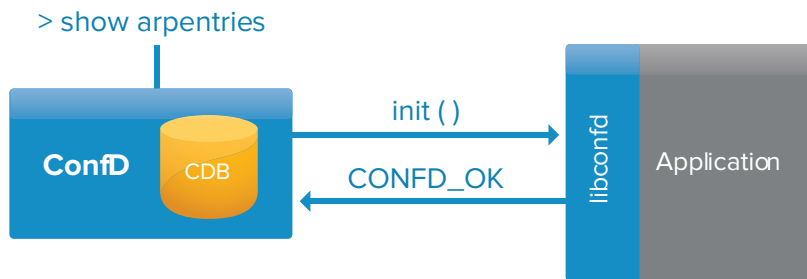


Figure 23: The control socket init message and response



Figure 24: The worker socket `get_elem` and `get_next` queries

ConfD may be configured to cache operational data provided by callpoints for a certain time. If an operator accesses such an element, its value will be taken from the cache, avoiding a call to the data provider.

### Storing Operational Data in CDB

An alternative approach to handling operational data is to store operational in CDB using a write interface. This is a suitable approach for long-lived operational data that is calculated by the application, e.g. performance data for the last 24 hours. This section is based on the `examples.conf/cdb_oper/ifstatus`.

Applications can write operational data to CDB. So instead of annotating the operational data with a callpoint, you state that the operational data should be managed by CDB (`tailf:cdb-oper`). In the YANG example below we have a list of interfaces and we state the counters are CDB operational data:

```

container interfaces {
  list interface {
    key name;
    max-elements 1024;
    leaf name {
      type string;
    }
    ...
    container status {
      config false;
      tailf:cdb-oper;

      container receive {
        leaf bytes {
          type uint64;
          mandatory true;
        }
      }
    }
  }
}

```

Figure 25: CDB Operational Data

You can control if the data is just kept in RAM or also made persistent to disk in journal files (`tailf:persistent`).

An application can maintain this data with the following steps:

1. `cdb_connect`, `cdb_start_session`
2. A series of calls to one or several of the CDB set functions
3. A call to `cdb_end_session`

It is also possible to load operational data from an XML file into CDB using the function `cdb_load_file`.

## Configuration Validation

ConfD stores device configuration data. Correct and consistent device configuration data is truly critical for the correct operations of the device. Misconfiguring a network device may lead to a situation where the device is no longer connected to the network. Before committing configuration data, it is crucial to ensure that the new configuration is reasonable.

Another benefit with a guaranteed validity of the configuration is that application software which reads the configuration data need not check the validity of the configuration.

ConfD has built-in support for several different types of validation. ConfD automatically checks that:

- **Syntactic validation:** the configuration data adheres to the YANG model types, ranges and structure.
- **Integrity constraints:** any uniqueness constraints are not violated and no YANG `leafref` references are left dangling.
- **Model constraints:** no YANG `must` or `when` statement expressions are violated. This is a very powerful mechanism whereby it's possible to instruct ConfD to compute an XPath expression that must always remain true, e.g. value X must be larger than the sum of all the numbers in the third column of list Y.
- **Application validation logic:** all applications that registered as validators approve the new configuration.
- **Operator policies:** any constraints specified by the operator are not violated, e.g. operator requires that encryption is always enabled on this device.

**Benefit:** guaranteed validity of the configuration is accomplished through no need for application software that reads the configuration data to check the validity of the configuration.

ConfD has built-in support for several different types of validation.

Let us look at a small example:

```
container notification {
  leaf protocol {
    type enumeration {
      enum SNMP;
      enum SMTP;
      enum NETCONF;
    }
    mandatory true;
  }
  leaf smtp-server {
    type inet:host;
  }
}
```

Figure 26: Validation Example Model

Assume that we want to make sure that if `protocol` is SMTP the `smtp-server` setting must not be empty. This could be done by having a `must` expression in the YANG model or referring to a validation-point.

```
// Using a must expression
container notification {
  leaf protocol {
    must "., != 'SMTP' or ../smtp-server" {
      error-message "Must specify smtp-server in order to
use SMTP protocol";
    }
    type enumeration {
...

// Using a validation-point
container notification {
  leaf protocol {
    tailf:validate smtp-app;
    type enumeration {
...

```

Figure 27: Must expression and validation-point

The `must` expression is of course preferred since it is declarative and model readers will understand the semantics of the model.

The architecture for implementing validation points is similar to what has been shown for callpoints previously.

## A System Example

In order to see how a little larger system with multiple independent applications could be organized, have a look at the `examples.conf.d/demo/quagga` example.

# The Bigger Picture of Network Management

## The Operator View

Network devices, virtual appliances and hosts must be configured and monitored. The management functions shall not assume human users; rather assume that the system is plugged into an overall *automated* system. Management functions must be exposed over user interfaces **and** management protocols/APIs to allow for automation. SDN and Network Automation has been an eye-opener for configuration management interfaces that are programmatic and *standardized* in contrast to only providing a CLI towards users.

The *configuration data* in the system must be *persistent* and be surrounded by *guard rails* so that only valid configurations can be entered. Configuration changes should not assume that users or systems know any *built-in sequencing* assumptions. Configuration changes should be done in an *ACID transaction*, all or nothing should be applied and manual rollback must be offered. The transactional behavior in combination with sequencing-independence makes it possible to have automated configuration applications.

Clear *separation of configuration data from operational data* is another old truth that is available in all good CLIs. It should be possible to get the complete configuration in one go, or filtered sub-trees thereof. Management interfaces that assume peek and poke of variables in general are extremely hard to use in any real scenarios.

Devices are installed and used in various scenarios; the above requirements need to be available over several interfaces:

- **NETCONF:** the IETF standard for device management allows transaction-safe programmability
- **REST:** for simple tasks reading and writing a few configuration and operational attributes
- **SNMP:** to support the vast majority of monitoring systems. Let NETCONF take care of writing configuration and have SNMP (and NETCONF) read operational data
- **CLI:** any device must support a CLI to power-users. The CLI must be highly user-friendly
- **Web UI:** a modern rich client Web UI that supports all tasks

## FCAPS

Fault-, Configuration-, Auditing-, Performance- and Security Management – these are the traditional areas of interest in a Network Operations Center (NOC). ConfD provides interfaces for building management systems encompassing all these aspects of device management.

## Network Automation and Software Defined Networking

When operators buy network equipment, typically around 80% of the Total Cost of Ownership (TCO) over 5 years goes to operational expenses (OPEX), i.e. running the system. Only 20% is the cost of hardware, capital expense (CAPEX).

Nearly half, 45% according to one study, of the OPEX is spent on configuring and provisioning services. That makes about one third of the operator's total spending go to configuration and provisioning – considerably more than the total spending on equipment.

Obviously, many operators are looking for ways to drastically reduce the OPEX. The easiest way is to drastically increase the network automation. Instead of having a cadre of people doing large parts of the configuration and provisioning work manually or semi-manually, service provisioning applications could do the work.

This is nothing new. The pressure to automate has been there for many, many years. Still, the OPEX improvements have been slow to materialize. This is because there have been a number of obstacles to efficiently implementing service provisioning applications.

Let's take an example. An operator would like to implement a service activation application for a new service. The network architects have designed how the service should work, with tunnels, servers, access control and billing. The necessary equipment has been installed and set up with a base configuration. Time to write the service activation application.

The traditional way of doing this has been to design a few web screens where the operator can fill in the service parameters, and then, when the operator hits the 'activate' button, a script is launched. This script inserts the provided parameters in the command stream and sends out CLI commands to some devices, TL1, SOAP or SNMP PDUs to others, etc. in some sequence.

The problem here is that the scripts, or device adapters in the more advanced solutions, quickly get expensive to develop and maintain. In fact, this has been the eternal headache for the entire networking business. The gap between network management system and network device has been too wide to efficiently bridge in a single layer.

This is where the Software Defined Networking (SDN) movement comes in. It brings a three-tier architecture for how network automation applications would communicate with network devices.

Key ideas in SDN is to separate the automation applications, a.k.a. service applications from the communications mechanisms employed to reach the devices and to leverage transactions as a mechanism to keep the complicated issues around error handling away from the automation applications.

The service application has an idea of what it would like the network setup to be. This idea needs to be conveyed to all the participating devices. To do so, traditionally it has been required to encode and serialize the idea in terms of CLI commands, SOAP messages, SNMP PDUs, etc. Each device then has to parse this serialization, and implement it.

From this perspective, any requirements that certain parameters must be specified on the same command line, or must be sent in separate PDUs, or that attribute X must be sent before attribute Y are just pointless bottlenecks in the flow of ideas from the manager to device.

### Key ideas in SDN

separate the automation applications, a.k.a. service applications from the communications mechanisms employed to reach the devices and to leverage transactions as a mechanism to keep the complicated issues around error handling away from the automation applications.



With a transactional interface, the network management application doesn't need to bother with this serialization, sequencing or error recovery. With a suitable encoding like NETCONF, the whole idea can also be transferred directly, without any need to break it down into individual commands.

Transaction support, and often times NETCONF and YANG support, are key requirements to make a device SDN ready.

### **NCS**

If you would like to understand more about Software Defined Networks (SDN) and the possible operator gains in efficiency, Tail-f Systems provides another product called Network Control System, NCS. You can find more information about NCS on the Tail-f website, [www.tail-f.com](http://www.tail-f.com).

# Evaluation Checklist

In this section you will find common questions that evaluators are seeking to answer when examining the ConfD solution. Feel free to use these questions for inspiration when looking for angles of review.

- Designer objectives: You will do the work, so your efficiency matters.
  - o Are you able to do everything you need to do?
  - o How soon can you show a new or updated feature to a customer?
  - o Is everything you need documented? Can you find what you need in the documentation, and is the description satisfactory?
  - o Is the product easy to use? Will you be efficient with this product?
- Operator objectives: You are doing all this for the operators. Will they be happy?
  - o Can operators manage everything using a single management system?
  - o Are you able to support 2-phase (or 3-phase) transactions?
  - o Are you able to support network-wide transactions?
  - o Are you able to support rollbacks, so operators can undo configuration changes in seconds, if needed?
  - o Can the kind of configuration validation you need be implemented with reasonable effort?
  - o CLI: Will the operator like the command line editing facilities, tab-completion, ?-help, history, search capabilities, output formatting, etc.? Are the prompts and show output formatted as operators expect? Is the responsiveness good? Can you tailor the CLI commands, look and feel as you need to?
  - o WebUI: Will the operator like the graphical appearance, the navigational aids, screen structure? Are the transactions, rollback, access control features implemented in a useful way? Can the WebUI be tailored to your needs with reasonable effort?
  - o NETCONF: Are all the mandatory and optional parts of NETCONF implemented well? Is the performance good? Can the operator achieve the network automation they are looking for?
  - o SNMP: Are the features operators are looking for implemented well? Is the performance in line with expectations?
  - o REST: Are the operator needs satisfied with the REST interface? Are the design principles of REST well implemented?
  - o Error handling: Are the error messages clear, precise and comprehensible by the average operator?

- Architecture: Bad architecture can be very expensive in the long run.
  - o Some features are hard to add afterwards, e.g. transactions. Are the necessary fundamental architectural components in place?
  - o Are the needs for High Availability, redundancy and database replication satisfied?
  - o Does the system conform to relevant standards?
  - o Are the requirements for online/offline software upgrade met?
  - o Can the system support your distributed system architecture well?
- Performance: Is the system fast and lean enough? Scales well?
  - o How long does it take to start the system when empty? With a large amount of data in the database?
  - o How long time does it take to commit a small or large configuration change? What about restoring a backup, where most of the data is the same as already in the database?
  - o What happens to the execution time of various operations when there is twice as much data in the system? Or 1,000 times more data? Or 100 concurrent operators?
  - o How much disk and RAM is needed?
  - o Is it stable, or are there crashes?
  - o Can it scale well on parallel CPUs and with multiple parallel workloads, e.g. many operators in parallel?
- Security: Transparent to authorized users, locking out everybody else.
  - o Authentication: Can you plug in the authentication sources you need? Can you build the right sequence of sources to consult?
  - o Authorization: Do you get the right level or authorization granularity and can you express the authorization rules you need? Is the performance good?
  - o Audit trails: Can you configure the system to produce the required audit trail information?
  - o Holes: Have you found any security holes?
  - o Ease of use: Security can be complicated, is this reasonably easy to use?
- Integration: It must fit with your existing system to be of any use. Does it fit?
  - o Fit with EMS/NMS/OSS system? What are the cost savings by allowing the management layer to use transactions?
  - o Fit with applications within the system?
  - o Are there integrations with 3rd party applications you use? E.g. Management agent functionality, operating systems, routing stacks, HA frameworks, ...?

## Summary

As far as we are told, most ConfD evaluators have come to the conclusion that there are really no viable product alternatives to ConfD. The only alternative that remains is to roll-your-own, i.e. it's a build-vs.-buy decision.

Things to consider when comparing ConfD with developing an in-house solution are elaborated below. Remember that ConfD has been tested in the field for several years in nearly 100 deployment scenarios.

- **Transaction management** taking all components into consideration: data-store, instrumentation and simultaneous requests over northbound APIs may be challenging to develop.
- End-user **requirements on the CLI**: network engineers expect the CLI to behave like a Cisco or Juniper style. The effort to achieve this is easily under-estimated.
- Supporting **several management APIs** with same feature set. Customers require programmatic APIs like NETCONF and REST, monitoring over SNMP, and CLI and WebUI user interfaces. Efficiently maintaining these across the feature set is non-trivial.
- **Core-features** like AAA, auditing, logs and session management are base features operators simply expect to be there.
- Effort to support **framework features** like configuration persistence and high-availability replication typically take several product generations to get right.

Using ConfD, the application team can focus on the unique application features and let ConfD provide market leading management capabilities.

## Further Information

- ConfD's User Guide: Look in the `doc/` directory of the ConfD installation
- ConfD's man pages: `man confd.conf`, etc. Look in the `man/` directory
- ConfD's Performance Data sheet: Your support contact can provide you with the latest version
- ConfD's examples collection: Look in the `examples.conf/` directory, there are about 80 example applications provided with ConfD
- The Tail-f website: [www.tail-f.com](http://www.tail-f.com)



**tail-f**

[www.tail-f.com](http://www.tail-f.com)

[info@tail-f.com](mailto:info@tail-f.com)

**Corporate Headquarters**

Korgmakargränd 2  
SE-111 22 Stockholm  
Sweden  
+46 8 21 37 40

**US Headquarters**

5201 Great American Pkwy  
Suite 320  
Santa Clara, CA 95054  
+1 408 466 4241

**Japanese Distributor**

A.I. Corporation  
Iijima Bldg., 2-25-2  
Nishi-Gotanda  
Shinagawa-ku  
Tokyo, 141-0031  
Japan  
+81 3 3493 7981