ConfD User Guide

ConfD 6.0

June 22, 2015

ConfD User Guide

ConfD 6.0

Publication date June 22, 2015

Table of Contents

List of Tables

List of Examples

Chapter 1. About the Documentation

1.1. How to Read This Guide

This document provides a wealth of information about ConfD and how to use it for your particular needs. It is written to be useful both when read front-to-back and also for readers that need to dive into particular aspects of the many features of ConfD.

Readers that are new to ConfD will learn a lot about how to think about, and apply, the features of ConfD by reading the first twelve chapters of this guide. They give an overview of the foundations of ConfD and how they can be used in various types of environments to meet various types of needs. Having read these chapters will also be useful as a guide during early design decisions to avoid missing out on useful ConfD features or applying features in a less than optimal way.

The rest of the document provides information about particular parts of ConfD. Time permitting, it is very useful to read as a whole, but they may also be read selectively depending on which parts of ConfD you are planning to use.

This document also consists of manual pages. The manual pages are reference information for the various tools, libraries and configuration files that are included in the ConfD package. They can also be found in native manual page format in the ConfD release package.

1.2. Getting Documentation

Updated documentation sets are prepared along with ConfD releases and can always be found in the customer download area or as part of the various deliverables. All releases contains the following updated documents:

- The *ConfD User Guide* is this document and is a separate download
- The *CHANGES file* describes all new features and corrections in the release and is a separate download
- The *HIGHLIGHTS* document is released with major releases and describes, with examples, all substantial new features per release. The HIGHLIGHTS document is a separate download.
- The *KNOWN_ISSUES* file is part of the release package and documents all known open issues at the time of release
- All ConfD release packages include a *README file* that describes how to install, set up and get started with ConfD. The README file is located in the top directory of a ConfD installation.
- The example collection includes a *README file* that introduces the reader to the wide selection of examples and what they contain. The README is contained in the examples deliverable.

All of the documents listed above contain information that is essential to the understanding of how to extract the most value out of ConfD and we urge all our users to read them.

1.3. Formatting Conventions

We use the following text and syntax conventions throughout the documentation:

Operating system references (e.g. commands, environment variables, filenames and command options) are rendered in `fixed-width font`

Programming language constructs (e.g. functions, constants and error codes) are rendered in `fixed-width font`

Multi-line code snippets and screen output are rendered like this:

```
# confdc -c test.cs
# confdc -l -o test.fxs test.xso
```

We use the following admonitions throughout this document:

Tip

This an example of a *tip* that is used to describe practical information on how to apply, or think about a certain aspect of the product

Note

This is an example of a *note* that is used to highlight a particular piece of information

Warning

This is an example of a *warning* that points out information that needs particular attention to avoid problems

1.4. Documentation Feedback

We appreciate documentation feedback, comments and suggestions so that we can continuously improve the documentation and make it more useful. Use the request tracker system to send us your comments and make sure you include information about which version and what section of the documentation you are referring to.

Chapter 2. An introduction to ConfD

2.1. An on-device software system for configuration management

Network devices, such as routers, switches or gateways, need to be configured and monitored. A fair amount of software is embedded in these devices to facilitate configuration and monitoring. This software typically includes:

- An SNMP agent for monitoring the device (SNMP is in practice almost never used for configuring devices, although it is possible to do so).
- Software to drive and render a command line interface (CLI).
- A small web server and content making up a device-specific web site, for a web-based user interface to the device management system.

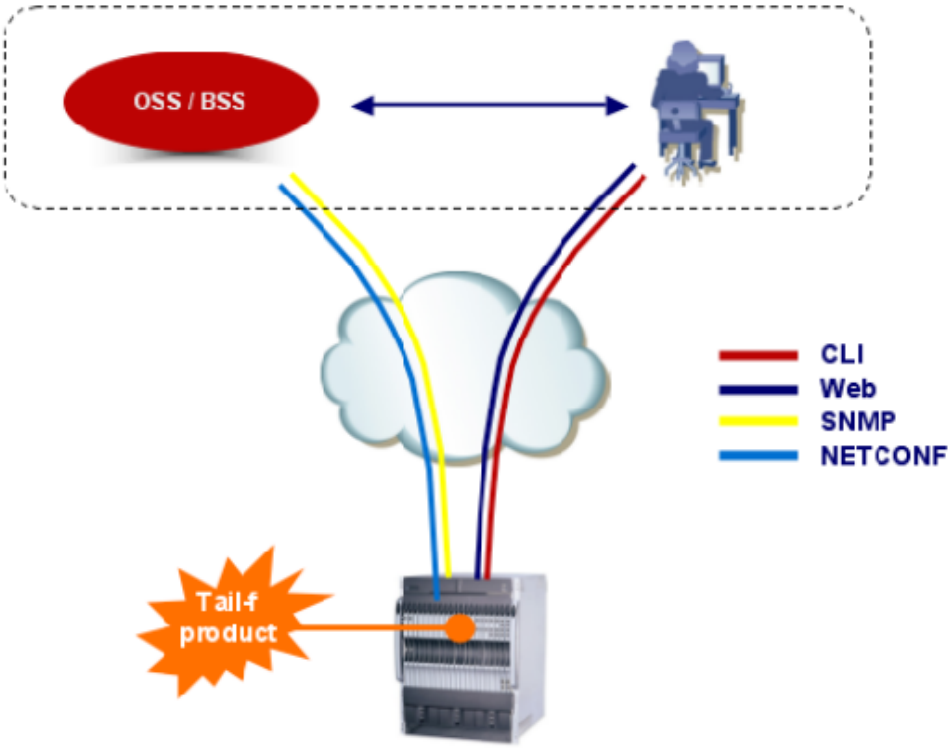
In addition, the IETF has developed a standard called NETCONF for automated configuration of network devices. NETCONF allows devices to expose an XML-based API that the network operator can use to set and get full and partial configuration data sets.

NETCONF solves several management problems that have been lacking standardized solutions. However, for an engineering organization with limited resources and a tight time schedule introducing/implementing NETCONF also poses a problem; a whole new management sub-system needs to be implemented and integrated with the other already existing management components, while time-to-market requirements remain unchanged.

2.2. ConfD Architecture

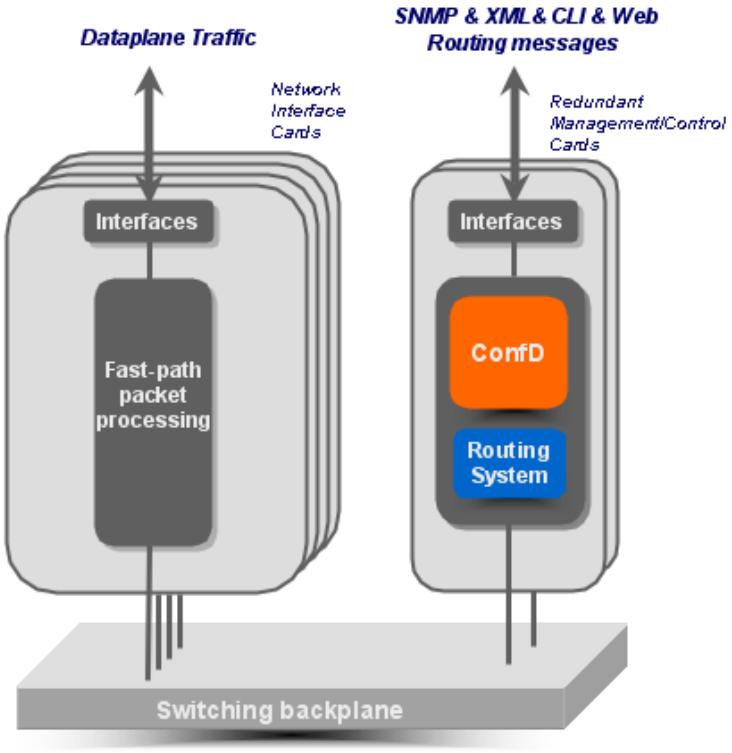
Tail-f's ConfD is a device configuration toolkit meant to be integrated as a management sub-system in network devices, providing:

- An implementation of the NETCONF protocol
- Automatic rendering of northbound interfaces, including CLI, Web UI and NETCONF
- Clustered/fault-tolerant storage of configuration data
- Master-agent/sub-agent framework for NETCONF, CLI, Web UI and SNMP



ConfD as sub-system on a network device

The following figure illustrates where ConfD would reside on, for example, a chassis-based router:



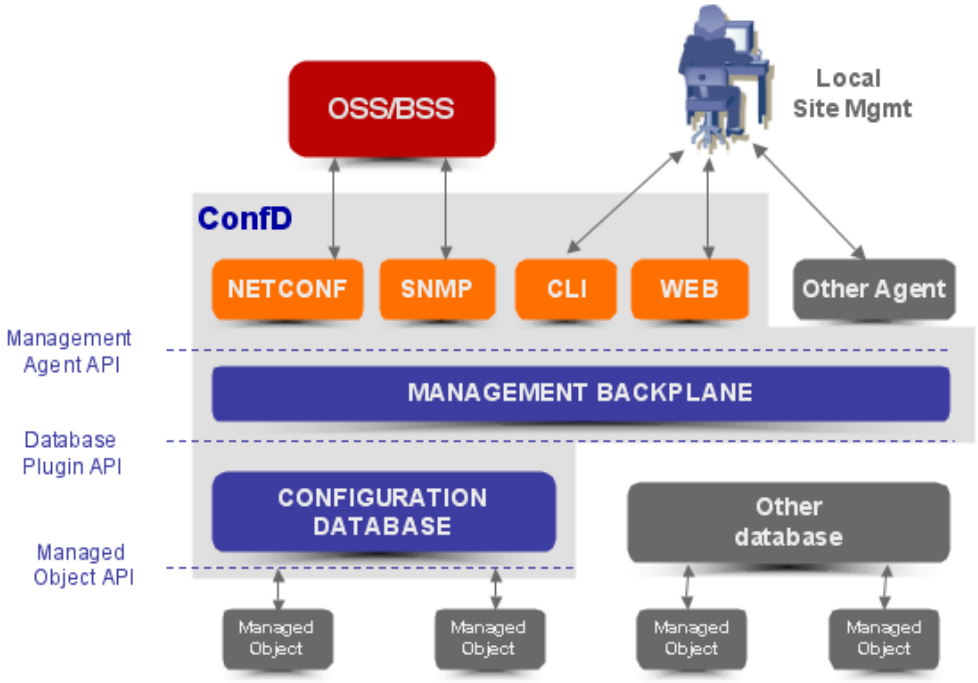
ConfD on a chassis-based router

ConfD executes as a regular Unix daemon on the target device, acting:

- as a NETCONF agent for the NETCONF protocol
- as a Web server for the Web UI
- as a CLI engine for command-line access
- and as an SNMP agent

It also contains a built-in XML configuration database.

The following figure illustrates the overall architecture. The ConfD architecture is modular, with well-defined interfaces between sub-systems.



ConfD architecture

The NETCONF, SNMP, CLI and Web modules are Management Agents. These communicate with external managers, and provide the managers with a protocol-specific view of the system. The box labeled Other Agent is e.g. a GUI application or some other management protocol implementation. These other Agents use the Management Agent API (MAAPI) to talk to the Management Backplane.

The Management Backplane provides an hierarchical view of the configuration and status/statistics data through the Management Agent API. This API is a session-oriented read/write API to the hierarchical data, with transaction-like semantics.

Examples of operations in this interface are 'create-subtree', 'get-instance', 'set-instance'. This interface is used both when the configuration is stored in the built-in ConfD database, and when it is stored in an external database.

The Management Backplane authenticates incoming requests through an AAA (Authentication, Authorization, Accounting) plugin API. An AAA plugin authenticates users and authorizes their requests. ConfD comes with a built-in AAA plugin, which can be replaced by vendor specific code.

In order to actually read and write the device-native configuration data, the sessions in the Management Backplane use the Database Plugin API. A database plugin has to provide mapping from the hierarchical view of the data used in the management protocols, to the native view used by the management database.

The management database can either be the integrated management database - called CDB - or some other database. CDB is a light-weight fault-tolerant distributed XML database. CDB can be used in single or multi-node systems in master slave configuration. It handles updates to the database schema automatically.

The Managed Objects in the application use the Managed Object API to read their configuration from the ConfD management database. There is a also a subscription mechanism, which the Managed Objects can use to react on configuration changes.

ConfD provides language bindings for the callback oriented plugin interfaces in C and Java. In the figure above, the Database Plugin API and the AAA Plugin API are available in C and Java The normal function call oriented APIs are available as C or Java APIs.

Chapter 3. The YANG Data Modeling Language

3.1. The YANG Data Modeling Language

YANG is a data modeling language used to model configuration and state data manipulated by a NETCONF agent. The YANG modeling language is defined in RFC 6020. YANG as a language will not be described in its entirety here - rather we refer to the IETF RFC text at http://www.ietf.org/rfc/rfc6020.txt.

Another source of information regarding the YANG language is the wiki based web site http://www.yang-central.org/. For a tutorial on the data modeling capabilities of YANG, see http://www.yang-central.org/twiki/bin/view/Main/DhcpTutorial.

3.2. YANG in ConfD

In ConfD , YANG is not only used for NETCONF data. On the contrary, YANG is used to describe the data model as a whole and used by all northbound interfaces.

A YANG module can be directly transformed into a final schema (.fxs) file that can be loaded into ConfD . Currently all features of the YANG language except the `anyxml` statement are supported.

3.3. YANG Introduction

This section is a brief introduction to YANG. The exact details of all language constructs is fully described in RFC 6020.

The ConfD programmer must know YANG well, since all APIs use various paths that are derived from the YANG datamodel.

3.3.1. Modules and Submodules

A module contains three types of statements: module-header statements, revision statements, and definition statements. The module header statements describe the module and give information about the module itself, the revision statements give information about the history of the module, and the definition statements are the body of the module where the data model is defined.

A module may be divided into submodules, based on the needs of the module owner. The external view remains that of a single module, regardless of the presence or size of its submodules.

The `include` statement allows a module or submodule to reference material in submodules, and the `import` statement allows references to material defined in other modules.

3.3.2. Data Modeling Basics

YANG defines four types of nodes for data modeling. In each of the following subsections, the example shows the YANG syntax as well as a corresponding NETCONF XML representation.

3.3.3. Leaf Nodes

A leaf node contains simple data like an integer or a string. It has exactly one value of a particular type, and no child nodes.

```
leaf host-name {
    type string;
    description "Hostname for this system";
}
```

With XML value representation for example as:

```
<host-name>my.example.com</host-name>
```

An interesting variant of leaf nodes are typeless leafs.

```
leaf enabled {
    type empty;
    description "Enable the interface";
}
```

With XML value representation for example as:

```
<enabled/>
```

3.3.4. Leaf-list Nodes

A `leaf-list` is a sequence of leaf nodes with exactly one value of a particular type per leaf.

```
leaf-list domain-search {
         type string;
         description "List of domain names to search";
     }
```

With XML value representation for example as:

```
<domain-search>high.example.com</domain-search>
<domain-search>low.example.com</domain-search>
<domain-search>everywhere.example.com</domain-search>
```

3.3.5. Container Nodes

A `container` node is used to group related nodes in a subtree. It has only child nodes and no value and may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).

```
container system {
    container login {
        leaf message {
            type string;
            description
                "Message given at start of login session";
        }
    }
}
```

With XML value representation for example as:

```
<system>
  <login>
    <message>Good morning, Dave</message>
```

```
  </login>
</system>
```

3.3.6. List Nodes

A `list` defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple keys and may contain any number of child nodes of any type (including leafs, lists, containers etc.).

```
list user {
    key "name";
    leaf name {
        type string;
    }
    leaf full-name {
        type string;
    }
    leaf class {
        type string;
    }
}
```

With XML value representation for example as:

```
<user>
  <name>glocks</name>
  <full-name>Goldie Locks</full-name>
  <class>intruder</class>
</user>
<user>
  <name>snowey</name>
  <full-name>Snow White</full-name>
  <class>free-loader</class>
</user>
<user>
  <name>rzull</name>
  <full-name>Repun Zell</full-name>
  <class>tower</class>
</user>
```

3.3.7. Example Module

These statements are combined to define the module:

```
// Contents of "acme-system.yang"
module acme-system {
    namespace "http://acme.example.com/system";
    prefix "acme";

    organization "ACME Inc.";
    contact "joe@acme.example.com";
    description
        "The module for entities implementing the ACME system.";

    revision 2007-06-09 {
```

```
        description "Initial revision.";
    }

    container system {
        leaf host-name {
            type string;
            description "Hostname for this system";
        }

        leaf-list domain-search {
            type string;
            description "List of domain names to search";
        }

        container login {
            leaf message {
                type string;
                description
                    "Message given at start of login session";
            }

            list user {
                key "name";
                leaf name {
                    type string;
                }
                leaf full-name {
                    type string;
                }
                leaf class {
                    type string;
                }
            }
        }
    }
}
```

3.3.8. State Data

YANG can model state data, as well as configuration data, based on the `config` statement. When a node is tagged with `config false`, its sub hierarchy is flagged as state data, to be reported using NETCONF's **get** operation, not the **get-config** operation. Parent containers, lists, and key leafs are reported also, giving the context for the state data.

In this example, two leafs are defined for each interface, a configured speed and an observed speed. The observed speed is not configuration, so it can be returned with NETCONF **get** operations, but not with **get-config** operations. The observed speed is not configuration data, and cannot be manipulated using **edit-config**.

```
list interface {
    key "name";
    config true;

    leaf name {
        type string;
    }
    leaf speed {
```

```
        type enumeration {
            enum 10m;
            enum 100m;
            enum auto;
        }
    }
    leaf observed-speed {
        type uint32;
        config false;
    }
}
```

3.3.9. Built-in Types

YANG has a set of built-in types, similar to those of many programming languages, but with some differences due to special requirements from the management domain. The following table summarizes the built-in types.

Table 3.1. YANG built-in types

Name	Type	Description
binary	Text	Any binary data
bits	Text/Number	A set of bits or flags
boolean	Text	"true" or "false"
decimal64	Number	64-bit fixed point real number
empty	Empty	A leaf that does not have any value
enumeration	Text/Number	Enumerated strings with associated numeric values
identityref	Text	A reference to an abstract identity
instance-identifier	Text	References a data tree node
int8	Number	8-bit signed integer
int16	Number	16-bit signed integer
int32	Number	32-bit signed integer
int64	Number	64-bit signed integer
leafref	Text/Number	A reference to a leaf instance
string	Text	Human readable string
uint8	Number	8-bit unsigned integer
uint16	Number	16-bit unsigned integer
uint32	Number	32-bit unsigned integer
uint64	Number	64-bit unsigned integer
union	Text/Number	Choice of member types

3.3.10. Derived Types (typedef)

YANG can define derived types from base types using the `typedef` statement. A base type can be either a built-in type or a derived type, allowing a hierarchy of derived types. A derived type can be used as the argument for the `type` statement.

```
typedef percent {
    type uint16 {
        range "0 .. 100";
    }
    description "Percentage";
}

leaf completed {
    type percent;
}
```

With XML value representation for example as:

```
<completed>20</completed>
```

User defined typedefs are useful when we want to name and reuse a type several times. It is also possible to restrict leafs inline in the data model as in:

```
leaf completed {
    type uint16 {
        range "0 .. 100";
    }
    description "Percentage";
}
```

3.3.11. Reusable Node Groups (grouping)

Groups of nodes can be assembled into the equivalent of complex types using the `grouping` statement. `grouping` defines a set of nodes that are instantiated with the `uses` statement:

```
grouping target {
    leaf address {
        type inet:ip-address;
        description "Target IP address";
    }
    leaf port {
        type inet:port-number;
        description "Target port number";
    }
}

container peer {
    container destination {
        uses target;
    }
}
```

With XML value representation for example as:

```
<peer>
  <destination>
    <address>192.0.2.1</address>
    <port>830</port>
  </destination>
</peer>
```

The grouping can be refined as it is used, allowing certain statements to be overridden. In this example, the description is refined:

```
container connection {
    container source {
        uses target {
            refine "address" {
                description "Source IP address";
            }
            refine "port" {
                description "Source port number";
            }
        }
    }
    container destination {
        uses target {
            refine "address" {
                description "Destination IP address";
            }
            refine "port" {
                description "Destination port number";
            }
        }
    }
}
```

3.3.12. Choices

YANG allows the data model to segregate incompatible nodes into distinct choices using the `choice` and `case` statements. The `choice` statement contains a set of `case` statements which define sets of schema nodes that cannot appear together. Each `case` may contain multiple nodes, but each node may appear in only one `case` under a `choice`.

When the nodes from one case are created, all nodes from all other cases are implicitly deleted. The device handles the enforcement of the constraint, preventing incompatibilities from existing in the configuration.

The choice and case nodes appear only in the schema tree, not in the data tree or XML encoding. The additional levels of hierarchy are not needed beyond the conceptual schema.

```
container food {
   choice snack {
       mandatory true;
       case sports-arena {
           leaf pretzel {
               type empty;
           }
           leaf beer {
               type empty;
           }
       }
       case late-night {
           leaf chocolate {
               type enumeration {
                   enum dark;
                   enum milk;
                   enum first-available;
               }
           }
```

```
        }
    }
}
```

With XML value reprentation for example as:

```
<food>
  <chocolate>first-available</chocolate>
</food>
```

3.3.13. Extending Data Models (augment)

YANG allows a module to insert additional nodes into data models, including both the current module (and its submodules) or an external module. This is useful e.g. for vendors to add vendor-specific parameters to standard data models in an interoperable way.

The `augment` statement defines the location in the data model hierarchy where new nodes are inserted, and the `when` statement defines the conditions when the new nodes are valid.

```
augment /system/login/user {
    when "class != 'wheel'";
    leaf uid {
        type uint16 {
            range "1000 .. 30000";
        }
    }
}
```

This example defines a `uid` node that only is valid when the user's `class` is not `wheel`.

If a module augments another model, the XML representation of the data will reflect the prefix of the augmenting model. For example, if the above augmentation were in a module with prefix `other`, the XML would look like:

```
<user>
  <name>alicew</name>
  <full-name>Alice N. Wonderland</full-name>
  <class>drop-out</class>
  <other:uid>1024</other:uid>
</user>
```

3.3.14. RPC Definitions

YANG allows the definition of NETCONF RPCs. The method names, input parameters and output parameters are modeled using YANG data definition statements.

```
rpc activate-software-image {
    input {
        leaf image-name {
            type string;
        }
    }
    output {
        leaf status {
            type string;
```

```
        }
    }
}
```

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <activate-software-image xmlns="http://acme.example.com/system">
    <name>acmefw-2.3</name>
 </activate-software-image>
</rpc>

<rpc-reply message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <status xmlns="http://acme.example.com/system">
    The image acmefw-2.3 is being installed.
  </status>
</rpc-reply>
```

3.3.15. Notification Definitions

YANG allows the definition of notifications suitable for NETCONF. YANG data definition statements are used to model the content of the notification.

```
notification link-failure {
    description "A link failure has been detected";
    leaf if-name {
        type leafref {
            path "/interfaces/interface/name";
        }
    }
    leaf if-admin-status {
        type ifAdminStatus;
    }
}
```

```
<notification xmlns="urn:ietf:params:netconf:capability:notification:1.0">
  <eventTime>2007-09-01T10:00:00Z</eventTime>
  <link-failure xmlns="http://acme.example.com/system">
    <if-name>so-1/2/3.0</if-name>
    <if-admin-status>up</if-admin-status>
  </link-failure>
</notification>
```

3.4. Working With YANG Modules

Assume we have a small trivial YANG file `test.yang`:

```
module test {
namespace "http://tail-f.com/test";
prefix "t";

  container top {
      leaf a {
          type int32;
      }
```

```
      leaf b {
          type string;
      }
  }
}
```

Tip

There is an Emacs mode suitable for YANG file editing in the system distribution. It is called `yang-mode.el`

We can use **confdc** compiler to compile the YANG module.

```
$ confdc -c test.yang
```

The above command creates an output file `test.fxs` that is a compiled schema that can be loaded into the system. The **confdc** compiler with all its flags is fully described in confdc (1).

There exists a number of standards based auxiliary YANG modules defining various useful data types. These modules, as well as their accompanying .fxs files can be found in the `${CONFD_DIR}/src/confd/yang` directory in the distribution.

The modules are:

- *ietf-yang-types* - defining some basic data types such as counters, dates and times.
- *ietf-inet-types* - defining several useful types related to IP addresses.

Whenever we wish to use any of those predefined modules we need to not only import the module into our YANG module, but we must also load the corresponding .fxs file for the imported module into the system.

So if we extend our test module so that it looks like:

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    import ietf-inet-types {
        prefix inet;
    }

    container top {
        leaf a {
            type int32;
        }
        leaf b {
            type string;
        }
        leaf ip {
            type inet:ipv4-address;
        }
    }
}
```

Normally when importing other YANG modules we must indicate through the `--yangpath` flag to **confdc** where to search for the imported module. In the special case of the standard modules, this is not required.

We compile the above as:

```
$ confdc -c test.yang
$ confdc --get-info test.fxs
fxs file
Confdc version:         "3.0_2" (current Confdc version = "3.0_2")
uri:                    http://tail-f.com/test
id:                     http://tail-f.com/test
prefix:                 "t"
flags:                  6
type:                   cs
mountpoint:             undefined
exported agents:        all
dependencies:           ['http://www.w3.org/2001/XMLSchema',
                         'urn:ietf:params:xml:ns:yang:inet-types']
source:                 ["test.yang"]
```

We see that the generated .fxs file has a dependency to the standard `urn:ietf:params:xml:ns:yang:inet-types` namespace. Thus if we try to start confd, we must also ensure that the fxs file for that namespace is loaded.

Failing to do so gives:

```
$ confd -c confd.conf --foreground --verbose
The namespace urn:ietf:params:xml:ns:yang:inet-types (referenced by http://tail-f.com/test)
Daemon died status=21
```

The remedy is to modify `confd.conf` so that it contains the proper load path or to provide the directory containing the fxs file, alternatively we can provide the path on the command line. The directory `${CONFD_DIR}/etc/confd` contains pre-compiled versions of the standard YANG modules.

```
$ confd -c confd.conf --addloadpath ${CONFD_DIR}/etc/confd --foreground --verbose
```

`confd.conf` is the configuration file for ConfD itself. It is described in confd.conf(5)

3.5. Integrity Constraints

The YANG language has built-in declarative constructs for common integrity constraints. These constructs are conveniently specified as `must` statements.

A `must` statement is an XPath expression that must evaluate to true or a non-empty node-set.

An example is:

```
container interface {
   leaf ifType {
       type enumeration {
           enum ethernet;
           enum atm;
       }
   }
   leaf ifMTU {
       type uint32;
```

```
    }
    must "ifType != 'ethernet' or "
      +  "(ifType = 'ethernet' and ifMTU = 1500)" {
        error-message "An ethernet MTU must be 1500";
    }
    must "ifType != 'atm' or "
       + "(ifType = 'atm' and ifMTU <= 17966 and ifMTU >= 64)" {
        error-message "An atm MTU must be  64 .. 17966";
    }
}
```

XPath is a very powerful tool here. It is often possible to express most realistic validation constraints using XPath expressions. Note that for performance reasons, it is recommended to use the `tailf:dependency` statement in the `must` statement. The compiler gives a warning if a `must` statement lacks a `tailf:dependency` statement, and it cannot derive the dependency from the expression. The options `--fail-on-warnings` or `-E TAILF_MUST_NEED_DEPENDENCY` can be given to force this warning to be treated as an error. See Section 9.9, “Dependencies - Why Does Validation Points Get Called” for details.

Another useful built-in constraint checker is the `unique` statement.

With the YANG code:

```
list server {
      key "name";
      unique "ip port";
      leaf name {
          type string;
      }
      leaf ip {
          type inet:ip-address;
      }
      leaf port {
          type inet:port-number;
      }
  }
```

We specify that the combination of IP and port must be unique. Thus the configuration:

```
<server>
  <name>smtp</name>
  <ip>192.0.2.1</ip>
  <port>25</port>
</server>

<server>
  <name>http</name>
  <ip>192.0.2.1</ip>
  <port>25</port>
</server>
```

is not valid.

The usage of leafrefs (See the YANG specification) ensures that we do not end up with configurations with dangling pointers. Leafrefs are also especially good, since the CLI and Web UI can render a better interface.

If other constraints are necessary, validation callback functions can be programmed in C. Read more about validation callbacks in Chapter 9, *Semantic validation*.

3.6. The when statement

The `when` statement is used to make its parent statement conditional. If the XPath expression specified as the argument to this statement evaluates to false, the parent node cannot be given configured. Furthermore, if the parent node exists, and some other node is changed so that the XPath expression becomes false, the parent node is automatically deleted. For example:

```
leaf a {
    type boolean;
}
leaf b {
    type string;
    when "../a = 'true'";
}
```

This data model snippet says that 'b' can only exist if 'a' is true. If 'a' is true, and 'b' has a value, and 'a' is set to false, 'b' will automatically be deleted.

Since the XPath expression in theory can refer to any node in the data tree, it has to be re-evaluated when any node in the tree is modified. But this would have a disastrous performance impact, so in order to avoid this, ConfD keeps track of dependencies for each when expression. In some simple cases, the **confdc** can figure out these dependencies by itself. In the example above, ConfD will detect that 'b' is dependant on 'a', and evaluate b's XPath expression only if 'a' is modified. If **confdc** cannot detect the dependencies by itself, it requires a `tailf:dependency` statement in the `when` statement. See Section 9.9, “Dependencies - Why Does Validation Points Get Called” for details.

3.7. Using the Tail-f Extensions with YANG

Tail-f has an extensive set of extensions to the YANG language that integrates YANG models in ConfD. For example when we have `config false;` data, we may wish to invoke user C code to deliver the statistics data in runtime. To do this we annotate the YANG model with a Tail-f extension called `tailf:callpoint`.

Alternatively we may wish to invoke user code to validate the configuration, this is also controlled through an extension called `tailf:validate`.

All these extensions are handled as normal YANG extensions. (YANG is designed to be extended) We have defined the Tail-f proprietary extensions in a file `${CONFD_DIR}/src/confd/yang/tailf-common.yang`

Continuing with our previous example, adding a callpoint and a validation point we get:

```
module test {
   namespace "http://tail-f.com/test";
   prefix "t";

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

```
      prefix tailf;
   }

   container top {
      leaf a {
          type int32;
          config false;
          tailf:callpoint mycp;
      }
      leaf b {
         tailf:validate myvalcp {
            tailf:dependency "../a";
         }
         type string;
      }
      leaf ip {
         type inet:ipv4-address;
      }
   }
}
```

The above module contains a callpoint and a validation point. The exact syntax for all Tail-f extensions are defined in the `tailf-common.yang` file.

Note the import statement where we import `tailf-common`.

When we are using YANG specifications in order to generate Java classes for ConfM, these extensions are ignored. They only make sense on the device side. It is worth mentioning them though, since EMS developers will certainly get the YANG specifications from the device developers, thus the YANG specifications may contain extensions

The man page tailf_yang_extensions(5) describes all the Tail-f YANG extensions.

3.7.1. Using a YANG annotation file

Sometimes it is convenient to specify all Tail-f extension statements in-line in the original YANG module. But in some cases, e.g. when implementing a standard YANG module, it is better to keep the Tail-f extension statements in a separate annotation file. When the YANG module is compiled to an fxs file, the compiler is given the original YANG module, and any number of annotation files.

A YANG annotation file is a normal YANG module which imports the module to annotate. Then the `tailf:annotate` statement is used to annotate nodes in the original module. For example, the module test above can be annotated like this:

```
module test {
   namespace "http://tail-f.com/test";
   prefix "t";

   import ietf-inet-types {
      prefix inet;
   }

   container top {
      leaf a {
          type int32;
          config false;
      }
```

```
      leaf b {
         type string;
      }
      leaf ip {
         type inet:ipv4-address;
      }
   }
}
```

```
module test-ann {
   namespace "http://tail-f.com/test-ann";
   prefix "ta";

   import test {
      prefix t;
   }
   import tailf-common {
      prefix tailf;
   }

   tailf:annotate "/t:top/t:a" {
       tailf:callpoint mycp;
   }

   tailf:annotate "/t:top" {
       tailf:annotate "t:b" {  // recursive annotation
           tailf:validate myvalcp {
               tailf:dependency "../t:a";
           }
       }
   }
}
```

In order to compile the module with annotations, use the -a parameter to **confdc**:

```
confdc -c -a test-ann.yang test.yang
```

3.8. Custom Help Texts and Error Messages

Certain parts of a YANG model are used by northbound agents, e.g. CLI and Web UI, to provide the end-user with custom help texts and error messages.

3.8.1. Custom Help Texts

A YANG statement can be annotated with a `description` statement which is used to describe the definition for a reader of the module. This text is often too long and too detailed to be useful as help text in a CLI. For this reason, ConfD by default does not use the text in the `description` for this purpose. Instead, a tail-f specific statement, `tailf:info` is used. It is recommended that the standard `description` statement contains a detailed description suitable for a module reader (e.g. NETCONF client or server implementor), and `tailf:info` contains a CLI help text.

As an alternative, ConfD can be instructed to use the text in the `description` statement also for CLI help text. See the option **--use-description** to confdc (1).

For example, CLI uses the help text to prompt for a value of this particular type. The CLI shows this information during tab/command completion or if the end-user explicitly asks for help using the ?-character. The behavior depends on the mode the CLI is running in.

The Web UI uses this information likewise to help the end-user.

The `mtu` definition below has been annotated to enrich the end-user experience:

```
leaf mtu {
    type uint16 {
        range "1 .. 1500";
    }
    description
       "MTU is the largest frame size that can be transmitted
        over the network. For example, an Ethernet MTU is 1,500
        bytes. Messages longer than the MTU must be divided
        into smaller frames.";
    tailf:info
       "largest frame size";
}
```

3.8.2. Custom Help Text in a Typedef

Alternatively, we could have provided the help text in a `typedef` statement as in:

```
 typedef mtuType {
    type uint16 {
        range "1 .. 1500";
    }
    description
        "MTU is the largest frame size that can be transmitted over the
         network. For example, an Ethernet MTU is 1,500
         bytes. Messages longer than the MTU must be
         divided into smaller frames.";
    tailf:info
       "largest frame size";
}

leaf mtu {
    type mtuType;
}
```

If there is an explicit help text attached to a leaf, it overrides the help text attached to the type.

3.8.3. Custom Error Messages

A statement can have an optional error-message statement. The north-bound agents, for example, the CLI uses this to inform the end-user about a provided value which is not of the correct type. If no custom error-message statement is available ConfD generates a built-in error message, e.g. "1505 is too large.".

All northbound agents use the extra information provided by an `error-message` statement.

The `typedef` statement below has been annotated to enrich the end-user experience when it comes to error information:

```
typedef mtuType {
   type uint32 {
       range "1..1500" {
           error-message
              "The MTU must be a positive number not "
```

```
            + "larger than 1500";
      }
   }
}
```

3.9. Hidden Data

It is sometimes useful to hide nodes from some of the northbound interfaces. The `tailf:export` statement, or the **--export** compile directive can be used to hide an entire module. It is recommended to use the `tailf:export` statement. More fine grained control can be attained with the optional `tailf:hidden` statement.

The `tailf:hidden` statement names a `hide group`. All nodes belonging to the same hide group are treated the same way as fas as being hidden or invisible. The hide group name `full` is given a special meaning. The `full` hide group is hidden from all northbound interfaces, not just user interfaces.

A related situation is when some nodes should be displayed to the user only when a certain condition is met. For example, the "ethernet" subtree should be displayed only when the type of an interface is "ethernet". This is covered in the subsection "Conditional display" below.

3.9.1. Fully Hidden Nodes

This is nodes that may be useful for the MOs, but should be hidden from all northbound interfaces. An example is the set of physical network interfaces on a device and their types. This is "static" data, i.e. it can't be changed by configuration, but it can vary between different models of a device that run the same software, and the device-specific data can be provided via init file or through MAAPI.

This type of data could also be realized via a separate module where `tailf:export` is used to limit the visibility, but being able to have some nodes in the data model hidden while others are not allows for greater flexibility - e.g. lists in the config data can have hidden child nodes, which get instantiated automatically along with the visible config nodes.

3.9.2. Hiding Nodes from User Interfaces

This is data that is fully visible to programmatic northbound interfaces such as NETCONF, but normally hidden from user interfaces such as CLI and Web UI. Examples are data used for experimental or end-customer-specific features, similar to hidden commands in the CLI but for data nodes.

A user interface may give access to this type of data (and even totally hidden data) if the user executes an `unhide` command identifying hide group that should be revealed. After this the nodes belonging to the hide group appear the same as unhidden data, i.e. they're included in tab completion, listed by `show` commands etc.

A hide group can only be `unhidden` if the group is listed in the `confd.conf`. This means that a hide group will be completely hidden to the user interfaces unless it has been explicitly allowed to be unhidden in `confd.conf`. A password can optionally be required to unhide a group.

```
  <hideGroup>
     <name>debug</name>
     <password>secret</password>
  </hideGroup>
```

3.9.3. Conditional display

Sometimes it is convenient to hide some CLI commands, or Web UI elements, when certain conditions on the configuration are met. A typical example is a "discriminated union". One leaf is the type of something, and depending on the value of this leaf, different containers are visible:

```
typedef interface-type {
    type enumeration {
        enum ethernet;
        enum atm;
        enum appletalk;
    }
}

leaf if-type {
    type interface-type;
}
container ethernet {
    ...
}
container atm {
    ...
}
container appletalk {
    ...
}
```

In this example, the "ethernet" container should be visible to the user only when the value of "if-type" is "ethernet".

This can be accomplished by using the `tailf:display-when` statement. It contains an XPath expression which specifies when the node should be displayed:

```
container ethernet {
    tailf:display-when "../if-type = 'ethernet'";
    ...
}
container atm {
    tailf:display-when "../if-type = 'atm'";
    ...
}
container appletalk {
    tailf:display-when "../if-type = 'appletalk'";
    ...
}
```

With this data model, the CLI behaves like this:

```
% show
if-type ethernet;
ethernet {
    ...         # data for ethernet
}
```

```
% set <tab>
Possible completions:
  ethernet  if-type

% set if-type atm

% set atm ...   # create atm data

% delete <tab>
Possible completions:
  ethernet  if-type        # NOTE: ethernet NOT present
```

3.10. An Example: Modeling a List of Interfaces

Say for example that we want to model the interface list on a Linux based device. Running the **ip link list** command reveals the type of information we have to model

```
$ /sbin/ip link list
1: eth0: <BROADCAST,MULTICAST,UP>; mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:12:3f:7d:b0:32 brd ff:ff:ff:ff:ff:ff
2: lo: <LOOPBACK,UP>; mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: dummy0: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether a6:17:b9:86:2c:04 brd ff:ff:ff:ff:ff:ff
```

and this is how we want to represent the above in XML:

```
<?xml version="1.0"?>
<config xmlns="http://example.com/ns/link">
  <links>
    <link>
      <name>eth0</name>
      <flags>
        <UP/>
        <BROADCAST/>
        <MULTICAST/>
      </flags>
      <addr>00:12:3f:7d:b0:32</addr>
      <brd>ff:ff:ff:ff:ff:ff</brd>
      <mtu>1500</mtu>
    </link>

    <link>
      <name>lo</name>
      <flags>
        <UP/>
        <LOOPBACK/>
      </flags>
      <addr>00:00:00:00:00:00</addr>
      <brd>00:00:00:00:00:00</brd>
      <mtu>16436</mtu>
    </link>
  </links>
</config>
```

An interface or a `link` has data associated with it. It also has a name, an obvious choice to use as the key - the data item which uniquely identifies an individual interface.

The structure of a YANG model is always a header, followed by type definitions, followed by the actual structure of the data. A YANG model for the interface list starts with a header:

```
module links {
    namespace "http://example.com/ns/links";
    prefix link;

    revision 2007-06-09 {
      description "Initial revision.";
    }
    ...
```

A number of datatype definitions may follow the YANG module header. Looking at the output from **/sbin/ip** we see that each interface has a number of boolean flags associated with it, e.g. `UP`, and `NOARP`.

One way to model a sequence of boolean flags is as a sequence of statements:

```
leaf UP {
    type boolean;
    default false;
}
leaf NOARP {
    type boolean;
    default false;
}
```

A better way is to model this as:

```
leaf UP {
    type empty;
}
leaf NOARP {
    type empty;
}
```

We could choose to group these leafs together into a grouping. This makes sense if we wish to use the same set of boolean flags in more than one place. We could thus create a named grouping such as:

```
grouping LinkFlags {
    leaf UP {
        type empty;
    }
    leaf NOARP {
        type empty;
    }
    leaf BROADCAST {
        type empty;
    }
    leaf MULTICAST {
        type empty;
    }
    leaf LOOPBACK {
        type empty;
    }
    leaf NOTRAILERS {
        type empty;
    }
```

```
}
```

The output from **/sbin/ip** also contains Ethernet MAC addresses. These are best represented by the `mac-address` type defined in the `ietf-yang-types.yang` file. The `mac-address` type is defined as:

```
typedef mac-address {
    type string {
        pattern '[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}';
    }
    description
       "The mac-address type represents an IEEE 802 MAC address.

       This type is in the value set and its semantics equivalent to
       the MacAddress textual convention of the SMIv2.";
    reference
      "IEEE 802: IEEE Standard for Local and Metropolitan Area
                 Networks: Overview and Architecture
       RFC 2579: Textual Conventions for SMIv2";
}
```

This defines a restriction on the string type, restricting values of the defined type "mac-address" to be strings adhering to the regular expression `[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}` Thus strings such as `a6:17:b9:86:2c:04` will be accepted.

Queue disciplines are associated with each device. They are typically used for bandwidth management. Another string restriction we could do is to define an enumeration of the different queue disciplines that can be attached to an interface.

We could write this as:

```
typedef QueueDisciplineType {
   type enumeration {
      enum pfifo_fast;
      enum noqueue;
      enum noop;
      enum htp;
   }
}
```

There are a large number of queue disciplines and we only list a few here. The example serves to show that using enumerations we can restrict the values of the data set in a way that ensures that data entered always is valid from a syntactical point of view.

Now that we have a number of usable datatypes, we continue with the actual data structure describing a list of interface entries:

```
container links {
    list link {
        key name;
        unique addr;
        max-elements 1024;
        leaf name {
            type string;
        }
        container flags {
            uses LinkFlags;
        }
```

```
        leaf addr {
            type yang:mac-address;
            mandatory true;
        }
        leaf brd {
            type yang:mac-address;
            mandatory true;
        }
        leaf qdisc {
            type QueueDisciplineType;
            mandatory true;
        }
        leaf qlen {
            type uint32;
            mandatory true;
        }
        leaf mtu {
            type uint32;
            mandatory true;
        }
    }
}
```

The `key` attribute on the leaf named "name" is important. It indicates that the leaf is the instance key for the list entry named "link". All the `link` leafs are guaranteed to have unique values for their `name` leafs due to the key declaration.

If one leaf alone does not uniquely identify an object, we can define multiple keys. At least one leaf *must* be an instance key - we cannot have lists without a key.

List entries are ordered and indexed according to the value of the key(s).

3.10.1. Modeling Relationships

A very common situation when modeling a device configuration is that we wish to model a relationship between two objects. This is achieved by means of the `leafref` statements. A `leafref` points to a child of a list entry which either is defined using a `key` or `unique` attribute.

The `leafref` statement can be used to express three flavors of relationships: *extensions*, *specializations* and *associations*. Below we exemplify this by extending the "link" example from above.

Firstly, assume we want to put/store the queue disciplines from the previous section in a separate container - not embedded inside the `links` container.

We then specify a separate container , containing all the queue disciplines which each refers to a specific `link` entry. This is written as:

```
container queueDisciplines {
    list queueDiscipline {
        key linkName;
        max-elements 1024;
        leaf linkName {
            type leafref {
                path "/config/links/link/name";
            }
        }
```

```
        leaf type {
            type QueueDisciplineType;
            mandatory true;
        }
        leaf length {
            type uint32;
        }
    }
}
```

The `linkName` statement is both an instance key of the `queueDiscipline` list, and at the same time refers to a specific `link` entry. This way we can extend the amount of configuration data associated with a specific `link` entry.

Secondly, assume we want to express a restriction or specialization on Ethernet `link` entries, e.g. it should be possible to restrict interface characteristics such as 10Mbps and half duplex.

We then specify a separate container, containing all the specializations which each refers to a specific `link`:

```
container linkLimitations {
    list LinkLimitation {
        key linkName;
        max-elements 1024;
        leaf linkName {
            type leafref {
                path "/config/links/link/name";
            }
        }
        container limitations {
            leaf only10Mbs { type boolean;}
            leaf onlyHalfDuplex { type boolean;}
        }
    }
}
```

The `linkName` leaf is both an instance key to the `linkLimitation` list, and at the same time refers to a specific `link` leaf. This way we can restrict or specialize a specific `link`.

Thirdly, assume we want to express that one of the `link` entries should be the default link. In that case we enforce an association between a non-dynamic `defaultLink` and a certain `link` entry:

```
leaf defaultLink {
    type leafref {
        path "/config/links/link/name";
    }
}
```

3.10.2. Ensuring Uniqueness

Key leafs are always unique. Sometimes we may wish to impose further restrictions on objects. For example, we can ensure that all `link` entries have a unique MAC address. This is achieved through the use of the `unique` statement:

```
container servers {
    list server {
```

```
        key name;
        unique "ip port";
        unique "index";
        max-elements 64;
        leaf name {
            type string;
        }
        leaf index {
            type uint32;
            mandatory true;
        }
        leaf ip {
            type inet:ip-address;
            mandatory true;
        }
        leaf port {
            type inet:port-number;
            mandatory true;
        }
    }
}
```

In this example we have two `unique` statements. These two groups ensure that each server has a unique index number as well as a unique ip and port pair.

3.10.3. Default Values

A leaf can have a static or dynamic default value. Static default values are defined with the `default` statement in the data model. For example:

```
leaf mtu {
    type int32;
    default 1500;
}
```

and:

```
leaf UP {
    type boolean;
    default true;
}
```

A dynamic default value means that the default value for the leaf is the value of some other leaf in the data model. This can be used to make the default values configurable by the user. Dynamic default values are defined using the `tailf:default-ref` statement. For example, suppose we want to make the MTU default value configurable:

```
container links {
    leaf mtu {
        type uint32;
    }
    list link {
        key name;
        leaf name {
            type string;
        }
```

```
        leaf mtu {
            type uint32;
            tailf:default-ref '../../mtu';
        }
    }
}
```

Now suppose we have the following data:

```
<links>
  <mtu>1000</mtu>
  <link>
    <name>eth0</name>
    <mtu>1500</mtu>
  </link>
  <link>
    <name>eth1</name>
  </link>
</links>
```

In the example above, link `eth0` has the mtu 1500, and link `eth1` has mtu 1000. Since `eth1` does not have a mtu value set, it defaults to the value of `../../mtu`, which is 1000 in this case.

Note

Whenever a leaf has a default value it implies that the leaf can be left out from the XML document, i.e. mandatory = false.

With the default value mechanism an old configuration can be used even after having added new settings.

Another example where default values are used is when a new instance is created. If all leafs within the instance have default values, these need not be specified in, for example, a NETCONF create operation.

3.10.4. The Final Interface YANG model

Here is the final interface YANG model with all constructs described above:

```
module links {
    namespace "http://example.com/ns/link";
    prefix link;

    import ietf-yang-types {
        prefix yang;
    }


    grouping LinkFlagsType {
        leaf UP {
            type empty;
        }
        leaf NOARP {
            type empty;
        }
        leaf BROADCAST {
            type empty;
        }
        leaf MULTICAST {
```

```
            type empty;
        }
        leaf LOOPBACK {
            type empty;
      }
        leaf NOTRAILERS {
            type empty;
        }
    }

    typedef QueueDisciplineType {
        type enumeration {
            enum pfifo_fast;
            enum noqueue;
            enum noop;
            enum htb;
        }
    }
    container config {
        container links {
            list link {
                key name;
                unique addr;
                max-elements 1024;
                leaf name {
                    type string;
                }
                container flags {
                    uses LinkFlagsType;
                }
                leaf addr {
                    type yang:mac-address;
                    mandatory true;
                }
                leaf brd {
                    type yang:mac-address;
                    mandatory true;
                }
                leaf mtu {
                    type uint32;
                    default 1500;
                }
            }
        }
        container queueDisciplines {
            list queueDiscipline {
                key linkName;
                max-elements 1024;
                leaf linkName {
                    type leafref {
                        path "/config/links/link/name";
                    }
                }
                leaf type {
                    type QueueDisciplineType;
                    mandatory true;
                }
                leaf length {
                    type uint32;
                }
```

```
            }
        }
        container linkLimitations {
            list linkLimitation {
                key linkName;
                leaf linkName {
                    type leafref {
                        path "/config/links/link/name";
                    }
                }
                container limitations {
                    leaf only10Mbps {
                        type boolean;
                        default false;
                    }
                    leaf onlyHalfDuplex {
                        type boolean;
                        default false;
                    }
                }
            }
        }
        container defaultLink {
            leaf linkName {
                type leafref {
                    path "/config/links/link/name";
                }
            }
        }
    }
}
```

If the above YANG file is saved on disk, as `links.yang`, we can compile and link it using the **confdc** compiler:

```
$ confdc -c links.yang
```

We now have a ready to use schema file named `links.fxs` on disk.

This is an example of an XML document adhering to the `links.yang` model:

```
<?xml version="1.0"?>
<config xmlns="http://example.com/ns/link">
  <links>
    <link>
      <name>eth0</name>
      <flags>
        <UP/>
        <BROADCAST/>
        <MULTICAST/>
      </flags>
      <addr>00:12:3f:7d:b0:32</addr>
      <brd>ff:ff:ff:ff:ff:ff</brd>
      <mtu>1500</mtu>
    </link>

    <link>
      <name>lo</name>
```

```
      <flags>
        <UP/>
        <LOOPBACK/>
      </flags>
      <addr>00:00:00:00:00:00</addr>
      <brd>00:00:00:00:00:00</brd>
      <mtu>16436</mtu>
    </link>
  </links>

  <queueDisciplines>
    <queueDiscipline>
      <linkName>eth0</linkName>
      <type>pfifo_fast</type>
      <length>1000</length>
    </queueDiscipline>

    <queueDiscipline>
      <linkName>lo</linkName>
      <type>noqueue</type>
    </queueDiscipline>
  </queueDisciplines>

  <linkLimitations>
    <linkLimitation>
      <linkName>eth0</linkName>
      <limitations>
        <only10Mbps>true</only10Mbps>
      </limitations>
    </linkLimitation>
  </linkLimitations>

  <defaultLink>
    <linkName>eth0</linkName>
  </defaultLink>
</config>
```

We can verify that the above XML document actually adheres to the YANG model by using the **-x** option to confdc (1)

```
$ confdc -f links.fxs -x links.xml
```

To actually run this example, we need to copy the compiled `links.fxs` to a directory where ConfD can find it.

3.11. More on leafrefs

A leafref is a used to model relationships in the data model, as described in Section 3.10.1, “Modeling Relationships”. In the simplest case, the leafreaf is a single leaf that references a single key in a list:

```
list host {
    key "name";
    leaf name {
        type string;
    }
```

```
    ...
}

leaf host-ref {
    type leafref {
        path "../host/name";
    }
}
```

But sometimes a list has more than one key, or we need to refer to a list entry within another list. Consider this example:

```
list host {
    key "name";
    leaf name {
        type string;
    }

    list server {
        key "ip port";
        leaf ip {
            type inet:ip-address;
        }
        leaf port {
            type inet:port-number;
        }
        ...
    }
}
```

If we want to refer to a specific server on a host, we must provide three values; the host name, the server ip and the server port. Using leafrefs, we can accomplish this by using three connected leafs:

```
leaf server-host {
    type leafref {
        path "/host/name";
    }
}
leaf server-ip {
    type leafref {
        path "/host[name=current()/../server-host]/server/ip";
    }
}
leaf server-port {
    type leafref {
        path "/host[name=current()/../server-host]"
           + "/server[ip=current()/../server-ip]/../port";
    }
}
```

The path specification for `server-ip` means the ip address of the server under the host with same name as specified in `server-host`.

The path specification for `server-port` means the port number of the server with the same ip as specified in `server-ip`, under the host with same name as specified in `server-host`.

This syntax quickly gets awkward and error prone. ConfD supports a shorthand syntax, by introducing an XPath function `deref()` (see the section called “XPATH FUNCTIONS”). Technically, this function follows a leafreaf value, and returns all nodes that the leafref refer to (typically just one). The example above can be written like this:

```
leaf server-host {
    type leafref {
        path "/host/name";
    }
}
leaf server-ip {
    type leafref {
        path "deref(../server-host)/../server/ip";
    }
}
leaf server-port {
    type leafref {
        path "deref(../server-ip)/../port";
    }
}
```

Note that using the `deref` function is syntactic sugar for the basic syntax. The translation between the two formats is trivial. Also note that `deref()` is an extension to YANG, and third party tools might not understand this syntax. In order to make sure that only plain YANG constructs are used in a module, the parameter `--strict-yang` can be given to **confdc -c**.

3.12. Using Multiple Namespaces

There are several reasons for supporting multiple configuration namespaces. Multiple namespaces can be used to group common datatypes and hierarchies to be used by other YANG models. Separate namespaces can be used to describe the configuration of unrelated sub-systems, i.e. to achieve strict configuration data model boundaries between these sub-systems.

As an example, `datatypes.yang` is a YANG module which defines a reusable data type.

```
module datatypes {
  namespace "http://example.com/ns/dt";
  prefix dt;

  grouping countersType {
     leaf recvBytes {
        type uint64;
        mandatory true;
     }
     leaf sentBytes {
        type uint64;
        mandatory true;
     }
  }
}
```

We compile and link `datatypes.yang` into a final schema file representing the `http://example.com/ns/dt` namespace:

```
$ confdc -c datatypes.yang
```

To reuse our user defined `countersType`, we must import the `datatypes` module.

```
module test {
    namespace "http://tail-f.com/test";
    prefix "t";

    import datatypes {
        prefix dt;
    }

    container stats {
        uses dt:countersType;
    }
}
```

When compiling this new module that refers to another module, we must indicate to **confdc** where to search for the imported module:

```
$ confdc -c test.yang --yangpath /path/to/dt
```

confdc also searches for referred modules in the colon (:) separated path defined by the environment variable `YANG_MODPATH` and . (dot) is implicitly included. Thus we can run **pyang** on `test.yang` and it will search for the `datatypes.yang` file and import it.

3.13. Module Names, Namespaces and Revisions

We have three different entities that define our configuration data.

- The module name. A system typically consists of several modules. In the future we also expect to see standard modules in a manner similar to how we have standard SNMP modules.

  It is highly recommended to have the vendor name embedded in the module name, similar to how vendors have their names in proprietary MIB s today.

  The module name is in no way exposed to north bound managers. I.e. it never shows up in the CLI or to a NETCONF manager, it is solely used for internal purposes.

- The XML namespace. A module defines a namespace. This is an important part of the module header. For example we have:

  ```
  module acme-system {
      namespace "http://acme.example.com/system";
      .....
  ```

  The namespace string must uniquely define the namespace. It is very important that once we have settled on a namespace we never change it. The namespace string should remain the same between revisions of a product. Do not embed revision information in the namespace string since that breaks manager side NETCONF scripts.

- The `revision` statement as in:

```
module acme-system {
    namespace "http://acme.example.com/system";
    prefix "acme";

    revision 2007-06-09;
    .....
```

The revision is exposed to a NETCONF manager in the capabilities sent from the agent to the NETCONF manager in the initial hello message. The fine details of revision management is being worked on in the IETF NETMOD working group and is not finalized at the time of this writing.

What is clear though, is that a manager should base its version decisions on the information in the revision string.

A capabilities reply from a NETCONF agent to the manager may look as:

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
  <capability>urn:ietf:params:netconf:base:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
  <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
  <capability>http://example.com/ns/link?revision=2007-06-09</capability>
  ....
```

where the revision information for the `http://example.com/ns/link` namespace is encoded as `?revision=2007-06-09` using standard URI notation.

When we change the data model for a namespace, it is recommended to change the revision statement, and to never make any changes to the data model that are backwards incompatible. This means that all leafs that are added must be either optional or have a default value. That way it is ensured that old NETCONF client code will continue to function on the new data model. Section 10 of RFC 6020 defines exactly what changes can be made to a data model in order to not break old NETCONF clients.

3.14. Hash Values and the id-value Statement

Internally and in the programmer APIs, ConfD uses integer values to represent YANG node names and the namespace URI. This conserves space and allows for more efficient comparisons (including `switch` statements) in the user application code. By default, **confdc** automatically computes a hash value for the namespace URI and for each string that is used as a node name.

Conflicts can occur in the mapping between strings and integer values - i.e. the initial assignment of integers to strings is unable to provide a unique, bi-directional mapping. Such conflicts are extremely rare (but possible) when the default hashing mechanism is used.

The conflicts are detected either by **confdc** or by the ConfD daemon when it loads the `.fxs` files.

If there are any conflicts reported they will pertain to XML tags (or the namespace URI),

There are two different cases:

- Two different strings mapped to the same integer. This is the classical hash conflict - extremely rare due to the high quality of the hash function used. The resolution is to use the `tailf:id-value` statement to assign a different integer to one of the strings - we should choose a number above 2^31+1 to avoid collisions with the generated hash values. To assign a different integer to the namespace URI, we need to use the **--ns-id-value** option with **confdc** rather than `tailf:id-value`.

- One string mapped to two different integers. This is even more rare than the previous case - it can only happen if a hash conflict was detected and avoided through the use of `tailf:id-value` on one of the strings, and that string also occurs somewhere else. The resolution is to use `tailf:id-value` (or **--ns-id-value**) to assign the same integer on both occurrences of the string. As in the previous case, we should choose a number above 2^31+1.

3.15. Migrating from Confspecs to YANG

There are some constructs in confspecs that are not possible to directly translate into YANG models. Probably the most problematic is how we, using confspecs, can mount pieces of configuration in one confspec file into a mount point in another confspec file. It is also possible to mount entire namespaces, .fxs files, into mount points in other namespaces. Neither is possible with the YANG language.

YANG has the submodule concept as a construct to divide a module. If the overall goal with confspec mounting is to let different developers work on specific files that are subsequently linked together, that same goal can be achieved in three different ways with YANG. The method described in Section 3.15.3, “Partitioning the Module Through Augments Statements” is probably the closest match for the confspec mounting functionality, but one of the others may be more appropriate or convenient to use.

3.15.1. Submodules and the Uses Statement

The strategy here is have a top module, that simply consists of a series of include statements, a series of `uses` statements, and a set of submodules that all define their respective groupings:

```
module system {
    namespace "http://tail-f.com/system";
    prefix "s";

    include system-sub;

    uses top;
    uses sys;
}
```

And then we can have arbitrary many sub modules.

```
submodule system-sub {

    belongs-to system {
        prefix s;
    }

    import ietf-inet-types {
        prefix inet;
    }

    grouping top {
        container top {
```

```
            leaf foo {
               type int32;
            }
            leaf addr{
               type inet:ip-address;
            }
        }
    }

    grouping sys {
        leaf bar {
           type int32;
        }
    }
}
```

3.15.2. Submodules and toplevel structures

The strategy here is have a top module, that simply consists of a series of include statements, and then each of the submodules define top level structures.

```
module system {
    namespace "http://tail-f.com/system";
    prefix "s";

    include system-sub;
    include system-sub2;
    .....
}
```

And then we can have arbitrary many sub modules.

```
submodule system-sub {

    belongs-to system {
        prefix s;
    }

    import ietf-inet-types {
        prefix inet;
    }

    container top {
       leaf foo {
           type int32;
       }
       leaf addr{
           type inet:ip-address;
       }
    }

    leaf bar {
        type int32;
    }

}
```

The `include` statement in the top module will ensure that the structures defined in the submodules(s) get expanded.

3.15.3. Partitioning the Module Through Augments Statements

First we need a toplevel file that includes all the submodules:

```
module system {
    namespace "http://tail-f.com/system";
    prefix "s";

    include system-common;
    include system-sub;

}
```

Following that we have one submodule that defines some shared data types, and most important, the structure of the entire data model as a skeleton.

```
submodule system-common {

    belongs-to system {
        prefix s;
    }

    import ietf-inet-types {
        prefix inet;
    }

    typedef myType {
       type int32;
    }

    container system {
        container chassis {
        }
        container stats {
        }
    }
}
```

And finally, we have one or more submodules that augment the structure defined in `system-common.yang`

```
submodule system-sub {

    belongs-to system {
        prefix s;
    }

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

```
    include system-common;

    augment "/system/chassis" {
        leaf foo {
           type int32;
        }
        leaf addr{
           type inet:ip-address;
        }
    }

    augment "/s:system/s:stats" {
        leaf bar {
           type int32;
        }
    }
}
```

3.15.4. Miscellaneous

The confspec language has a construct called `indexedView`. This is a means to indicate in the data model that items in a list have an order. Some of the north bound agents can then insert new list entries given by the integer key values. Typical examples are things like firewall rules.

The `indexedView` concept is supported by the `tailf:indexed-view` extension to YANG. However in most cases, it will probably be more appropriate to use the standard, more powerful YANG mechanism called `ordered-by user`. Thus confspec users that use the `indexedView` may want to migrate to `ordered-by user` instead of `tailf:indexed-view`. Read more about `ordered-by user` in the YANG specification.

An alternative is to continue to use the Tail-f proprietary `indexedView` feature in the YANG model.

Yet another area that differs slightly are confspec `keyref`s. A confspec `keyref` uses tagpaths to indicate what it points to. The YANG equivalent is `leafref` that behave in a similar way. The major difference is that a `leafref` use an XPath expression instead of a tagpath.

Leafrefs that are not also keys may be on the tagpath form we have in confspecs, but they may also be instantiated XPath expressions - thus making leafrefs more powerful that confspec keyrefs.

We may also have list entries where one or more of the keys are leafrefs. As long as we have list entries with a single key there is no difference at all compared to confspecs. It is only when we have multiple keys that we see a difference. See Section 3.11, “More on leafrefs” for more info.

3.15.5. Enumerations

confspecs have enumeration on the form of:

```
<xs:simpleType name="YesNo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>
```

The equivalent enumeration in YANG looks as:

```
typedef YesNo {
    type enumeration {
       enum yes;
       enum no;
    }
}
```

These two enumerations are however not completely equivalent. YANG enumerations are implicitly numbered, starting at zero. Thus the exactly equivalent confspec enumerations looks like:

```
<xs:simpleType name="YesNo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="yes" idValue="0"/>
    <xs:enumeration value="no" idValue="1"/>
  </xs:restriction>
</xs:simpleType>
```

Thus, applications that are currently not using the **--allow-enum-conflicts** flag to **confdc** may break. The C API functions to map from hash value to string doesn't work properly for enumerations any longer when we use YANG models. When we use YANG models, the flag **--allow-enum-conflicts** is implicitly used due to the above mentioned fact that YANG enumerations always start at zero.

3.15.6. The Namespace URI

Prior to YANG we have recommended confspec users to embed version information in the string that identifies a namespace as in:

```
<confspec xmlns="http://tail-f.com/ns/confspec/1.0"
          xmlns:confd="http://tail-f.com/ns/confd/1.0"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://example.com/ns/interfaces/1.1"
          mount="/">
```

using for example the `targetNamespace` attribute. This is no longer recommended practice. The recommendation is now to never embed version information in the namespace URI, see Section 3.13, “Module Names, Namespaces and Revisions”.

Using the new naming scheme one would translate the example above into a YANG module like this:

```
module interfaces {
  namespace "http://example.com/ns/interfaces";
  prefix interfaces;
...
}
```

However if we need to support upgrade from a software release that used confspec to a new release that uses YANG for the same config we have to be careful. In confspec the identifier of a module is the part of the namespace before the version number. That is, in `http://example.com/ns/interfaces/1.1` the identifier is `http://example.com/ns/interfaces` (in confspec it can also be explicitly set using the `id` attribute on the top-level `confspec` node). This matters to CDB upgrade, in that CDB will use the *identifier* to determine if a namespace is the same or not, when upgrading (so that it can correctly identify `http://example.com/ns/interfaces/2.0` as the new version of `http://example.com/ns/interfaces/1.1`). When using YANG ConfD (and CDB) considers the whole namespace string (since it isn't supposed to change between revisions) as the identifier. To help there is a YANG extension called `tailf:id` which takes either an empty string (which means “treat the namespace

just as confspec would, and deduce the id from the namespace string") or an explicit string (the equivalent of using `id` in confspec).

In summary: if we need to support upgrade from a previous release *and* need to keep the exact namespace string we had in confspec we can translate the initial confspec fragment above to:

```
module interfaces {
  namespace "http://example.com/ns/interfaces/1.1";
  prefix interfaces;
  tailf:id "";
...
}
```

3.15.7. The cs2yang Tool

Finally we must mention the cs2yang tool. It is an XSLT based tool that can be used to translate confspec files into YANG files. There are some constructs that are not directly translatable, but the output from cs2yang provides an excellent starting point for migration.

Thus the first step in a migration from confspecs to YANG is to use the **cs2yang** tool on all confspec files we wish to translate.

Following that, we must decide on a multi-file mount strategy as described above. The generated files must then subsequently be hand edited into a suitable form.

In many confspec models, the namespace URI contains a version number. In YANG, the namespace URI of a data model must not change if the model is updated. Instead, YANG has a revision statement to handle data model versioning. Unless confspec based systems need to be upgraded in the field to YANG, it is recommended that the version number in the namespace URI is removed, and a revision statement is added. In this case, the `tailf:id` statement that cs2yang generated, must also be removed from the YANG module. See Section 3.15.6, "The Namespace URI" for a more detailed discussion of this problem.

3.16. The pyang tool

`pyang` is a Python based tool that is integrated into the `confdc` tool chain. Pyang can be used to check and transform YANG models. It has an extensible plugin architecture that allows users to plugin their own output modules. Thus if we wish to to transform YANG data models - processing e.g proprietary extensions, writing a Python plugin to pyang is a feasible way forward.

Pyang is developed by Tail-f, and it is open source. Code and documentation are available at http://www.yang-central.org/.

Pyang is also described in the man page pyang(1)

Chapter 4. Rendering Agents

4.1. Introduction

In this chapter we reintroduce the `links.yang` model from the Yang chapter and see how we can use that model to render all northbound interfaces.

This chapter is an overview, and all concepts touched upon in this chapter are thoroughly described in later chapters.

4.2. Data Model

The `links.yang` data model introduced in the "Yang" chapter defines a set of interfaces. Compiling the `links.yang` file into an .fxs file is the first required step.

```
module links {
    namespace "http://example.com/ns/link";
    prefix link;

    import ietf-yang-types {
        prefix yang;
    }


    grouping LinkFlagsType {
        leaf UP {
            type empty;
        }
        leaf NOARP {
            type empty;
        }
        leaf BROADCAST {
            type empty;
        }
        leaf MULTICAST {
            type empty;
        }
        leaf LOOPBACK {
            type empty;
      }
        leaf NOTRAILERS {
            type empty;
        }
    }

    typedef QueueDisciplineType {
        type enumeration {
            enum pfifo_fast;
            enum noqueue;
            enum noop;
            enum htb;
        }
    }
    container config {
        container links {
```

```
            list link {
                key name;
                unique addr;
                max-elements 1024;
                leaf name {
                    type string;
                }
                container flags {
                    uses LinkFlagsType;
                }
                leaf addr {
                    type yang:mac-address;
                    mandatory true;
                }
                leaf brd {
                    type yang:mac-address;
                    mandatory true;
                }
                leaf mtu {
                    type uint32;
                    default 1500;
                }
            }
        }
        container queueDisciplines {
            list queueDiscipline {
                key linkName;
                max-elements 1024;
                leaf linkName {
                    type leafref {
                        path "/config/links/link/name";
                    }
                }
                leaf type {
                    type QueueDisciplineType;
                    mandatory true;
                }
                leaf length {
                    type uint32;
                }
            }
        }
        container linkLimitations {
            list linkLimitation {
                key linkName;
                leaf linkName {
                    type leafref {
                        path "/config/links/link/name";
                    }
                }
                container limitations {
                    leaf only10Mbps {
                        type boolean;
                        default false;
                    }
                    leaf onlyHalfDuplex {
                        type boolean;
                        default false;
                    }
                }
```

```
            }
        }
        container defaultLink {
            leaf linkName {
                type leafref {
                    path "/config/links/link/name";
                }
            }
        }
    }
}
```

The command to compile the YANG file is **confdc -c links.yang** Thus in our Makefile we have:

```
all:   links.fxs

%.fxs:  %.yang
  $(CONFD_DIR)/bin/confdc -c $*.yang
```

The Makefile requires the UNIX environment variable `CONFD_DIR` to be set to the directory where ConfD is installed.

Once we have the .fxs file, it's time to start ConfD. To do that we have some initial steps that must be taken care of first.

We must have a configuration file for ConfD it self. This file is usually referred to as `confd.conf`. There is a multitude of things we can configure in confd.conf(5) but for this initial example we'll just focus on the things we need to get the `links.yang` example to work. We can copy the `confd.conf` file from `etc/confd/confd.conf` relative to the installation directory of ConfD and add "." to the `loadPath`

Now with our newly compiled `links.yang` file we can start ConfD as:

```
# source /path/to/installed_confd/confdrc
# confd --foreground --verbose -c ./confd.conf
```

This starts ConfD in the foreground with all log messages displayed on stdout. This is a convenient way of running ConfD during development.

4.3. Using the CLIs

The first thing we can try out is how a Juniper style CLI looks and feels on our data model. The ConfD CLI(s) are entirely rendered from the data model. It's possible to extend and tweak the looks of the CLI in several ways, but for now, we try just what we get from the plain data model. The ConfD CLI is run through a small C program called **confd_cli** which is typically used as a login shell on the target host. During development, it's usually more convenient to just invoke that program directly from the UNIX prompt. The **confd_cli** program communicates with the ConfD daemon over the loopback socket.

Here is an actual session:

```
# confd_cli
```

```
Welcome to the ConfD CLI
admin connected from 127.0.0.1 using console on buzz
admin@buzz 17:37:17> configure
Entering configuration mode private
[ok][2009-03-17 17:37:26]

[edit]
admin@buzz 17:37:26% set config links link eth0 addr 00:12:3f:7d:b0:32 brd 00:1
2:3f:7d:b0:32
[ok][2009-03-17 17:37:48]

[edit]
admin@buzz 17:37:48% set config links link eth0 flags BROADCAST
[ok][2009-03-17 17:38:05]

[edit]
admin@buzz 17:38:05% set config links link eth0 flags
Possible completions:
  BROADCAST  LOOPBACK  MULTICAST  NOARP  NOTRAILERS  UP
admin@buzz 17:38:05% set config links link eth0 flags LOOPBACK
[ok][2009-03-17 17:38:25]

[edit]
admin@buzz 17:38:25% commit
Commit complete.
[ok][2009-03-17 17:38:31]

[edit]
admin@buzz 17:38:31% exit
[ok][2009-03-17 17:38:34]
admin@buzz 17:38:34> show configuration config
links {
    link eth0 {
        flags {
            UP;
            BROADCAST;
            LOOPBACK;
        }
        addr 00:12:3f:7d:b0:32;
        brd  00:12:3f:7d:b0:32;
    }
}
[ok][2009-03-17 17:38:44]
admin@buzz 17:38:44> exit
#
```

Thus from the data model we got a fully functional Juniper like CLI.

We can also get a Cisco like CLI towards the same data model:

```
# confd_cli -C
Welcome to the ConfD CLI
admin connected from 127.0.0.1 using console on buzz
buzz# show running-config config
config links link eth0
 flags UP
 flags BROADCAST
 flags LOOPBACK
 addr 00:12:3f:7d:b0:32
```

```
 brd  00:12:3f:7d:b0:32
!
buzz# config
Entering configuration mode terminal
buzz(config)# config links link <TAB>
Possible completions:
  <name:string>  eth0
buzz(config)# config links link eth0 <TAB>
Possible completions:
  addr  brd  flags  mtu  <cr>
buzz(config)# config links link eth0 mtu 1200
buzz(config-link-eth0)# commit
Commit complete.
buzz(config-link-eth0)#
buzz(config)# config <TAB>
Possible completions:
  defaultLink  linkLimitations  links  queueDisciplines
buzz(config)# config defaultLink linkName eth0
buzz(config)# commit
Commit complete.
```

The rendered CLIs are highly capable containing all features expected by a modern CLI.

The data model contains a top level `config` container. This makes sense from a data modeling perspective since it wraps all configuration items inside a container. However, we may wish to do away with the container in the CLI. We want to issue the command `defaultLink linkName eth0` as opposed to the command `config defaultLink linkName eth0`.

The ConfD CLI can be tweaked in a myriad different ways. The typical development cycle is to define the data model to be as succinct and understandable as possible. This makes life easier for the application programmers who write C/C++ code that access the data model. Once the YANG model is good, we tweak the CLI to become what we want.

In this tiny example we write a really small CLI modification file:

```
<clispec xmlns="http://tail-f.com/ns/clispec/1.0" style="c">
  <operationalMode>
  </operationalMode>
  <configureMode>
    <modifications>
      <dropElem src="config"/>
    </modifications>
  </configureMode>
</clispec>
```

Save that file as `mods.cli` and compile that file as:

```
# confdc -c mods.cli
```

This results in a file called `mods.ccl` that needs to be put in the load path of ConfD. We then re-launch the Cisco CLI:

```
```

```
# confd_cli -C
Welcome to the ConfD CLI
admin connected from 127.0.0.1 using console on buzz
buzz# config
Entering configuration mode terminal
buzz(config)# links link eth0 <TAB>
Possible completions:
  addr  brd  flags  mtu  <cr>
buzz(config)# links link eth0 mtu ?
Possible completions:
  <unsignedInt>[1200]
buzz(config)# links link eth0 mtu 1400
buzz(config-link-eth0)# commit
Commit complete.
buzz(config-link-eth0)#
```

4.4. Using NETCONF

NETCONF is a powerful protocol that can be used to programmatically reconfigure a device. We're still running ConfD with the `links.yang` data model.

The easiest way to interact with the NETCONF agent in ConfD is to use a small python based program called **netconf-console** that ships with ConfD. Let's just run it from the UNIX prompt and see what we get:

```
# netconf-console --user=admin --password=admin --get-config -x '/config'
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <config xmlns="http://example.com/ns/link">
      <links>
        <link>
          <name>eth0</name>
          <flags>
            <UP/>
            <BROADCAST/>
            <LOOPBACK/>
          </flags>
          <addr>00:12:3f:7d:b0:32</addr>
          <brd>00:12:3f:7d:b0:32</brd>
          <mtu>1400</mtu>
        </link>
      </links>
      <queueDisciplines/>
      <linkLimitations>
        <linkLimitation>
          <linkName>eth0</linkName>
          <limitations>
            <only10Mbps>true</only10Mbps>
          </limitations>
        </linkLimitation>
      </linkLimitations>
      <defaultLink>
        <linkName>eth0</linkName>
      </defaultLink>
    </config>
  </data>
```

```
</rpc-reply>
```

The above command uses the python paramiko ssh client to establish an SSH session to ConfD. It then issues a NETCONF **get-config** RPC to retrieve all configuration data found below the XPath `/config`, i.e. this is precisely the data we have just entered in the CLI. The command **netconf-console --get-config** is a great way to extract a backup of the entire configuration of the device.

If we want to manipulate the configuration with the **netconf-console**program , we must prepare some XML data that can be sent as an **edit-config** and feed it to **netconf-console**.

Here is an example, save the following to a file, `set-mtu.xml`.

```
<edit-config>
  <target>
    <running/>
  </target>
  <config xmlns="http://example.com/ns/link">
    <config>
      <links>
        <link>
          <name>eth0</name>
          <mtu>1100</mtu>
        </link>
      </links>
    </config>
  </config>
</edit-config>
```

Then run this with:

```
# netconf-console --user=admin --password=admin --rpc set-mtu.xml
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <ok/>
</rpc-reply>
```

It's also fairly instructive to directly connect to the agent using OpenSSH as in:

```
# ssh -s -p 2022 admin@localhost netconf
admin@localhost's password:
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
<capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
<capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
<capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
<capability>urn:ietf:params:netconf:capability:url:1.0?scheme=ftp,file</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
<capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
<capability>http://example.com/ns/link</capability>
<capability>http://tail-f.com/ns/aaa/1.1</capability>
</capabilities>
<session-id>14</session-id></hello>
```

The agent replies with the capabilities it supports.

4.4.1. Generating Java classes for JNC

It is of course entirely possible to use the **netconf-console** program to XML script towards a NETCONF agent. An alternative is to use a Java library such as JNC to interact with the NETCONF agent. JNC is in the Open SOurce and can by found on github at `https://github.com/tail-f-systems/JNC/`.

4.5. Generating the Web UI

Similar to how we, starting with only the data model, render a CLI or a NETCONF agent, we can use a similar technique to render a modern AJAX based Web UI.

Just enabling the Web UI in confd.conf, and restarting ConfD gives us a plain Web UI. How to enable the Web UI with and without SSL support is described in confd.conf(5).

Chapter 5. CDB - The ConfD XML Database

5.1. Introduction

This chapter describes how to use ConfD's built-in configuration database CDB. As a running example, we will describe a DHCP daemon configuration. CDB can also be used to store operational data - read more about this in Section 6.8, “Operational data in CDB”.

A network device needs to store its configuration somewhere. Usually the device configuration is stored in a database or in plain files, sometimes a combination of both.

ConfD has a built-in XML database which can be used to store the configuration data for the device. The database is called CDB - Configuration DataBase.

5.2. CDB

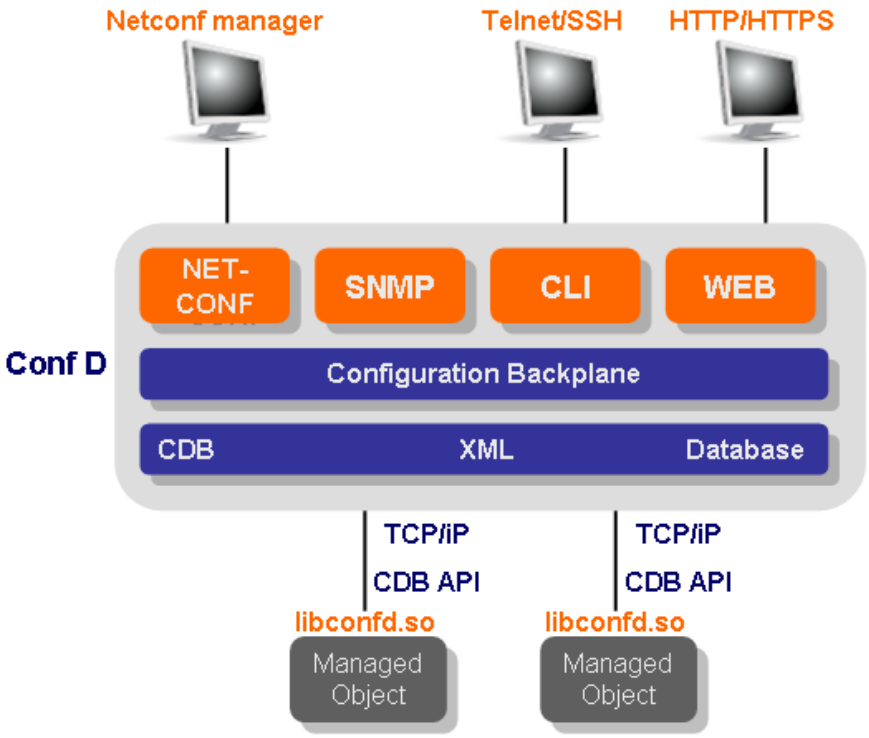
By default, ConfD stores all configuration data in CDB. The alternative is to use an external database as described in Chapter 7, *The external database API*. There are a number of advantages to CDB compared to using some external storage for configuration data. CDB has:

- A solid model on how to handle configuration data in network devices, including a good update subscription mechanism.
- A networked API whereby it is possible for an unconfigured device to find the configuration data on the network and use that configuration.
- Fast lightweight database access. CDB by default keeps the entire configuration in RAM as well as on disk.
- Ease of use. CDB is already integrated into ConfD, the database is lightweight and has no maintenance needs. Writing instrumentation functions to access data is easy.
- Automatic support for upgrade and downgrade of configuration data. This is a key feature, which is useful not only when performing actual up/downgrades on the device. It also greatly simplifies the development process by allowing individual developers to add/delete items in the configuration without any impact whatsoever on other developers. This will be fully described later.

When using CDB to store the configuration data, the applications need to be able to:

1. Read configuration data from the database.
2. React when the database is written to. There are several possible writers to the database, such as the CLI, NETCONF sessions, the Web UI, or the NETCONF agent. Suppose an operator runs the CLI and changes the value of some leaf. When this happens, the application needs to be informed about the configuration change.

The following figure illustrates the architecture when CDB is used.



ConfD CDB architecture scenario

The Applications/Managed Objects in the figure above read configuration data and subscribe to changes to the database using a simple RPC-based API. The API is part of the `libconfd.so` shared library and is fully documented in the UNIX man page confd_lib_cdb(3). Since the API is RPC-based, the Applications may run on other hosts that are not running ConfD - which could be used for example in a chassis-based system where ConfD only would run on the management blade, and the managed applications on other blades in the system.

5.3. An example

Let us look at a simple example which will illustrate how to populate the database, how to read from it using the C API, as well as react to changes to the data. First we need a YANG module (see Chapter 3, *The YANG Data Modeling Language* for more details about how to write data models in YANG). Consider this simplified, but functional, example:

Example 5.1. a simple server data model, `servers.yang`

```
module servers {
  namespace "http://example.com/ns/servers";
  prefix servers;

  import ietf-inet-types {
    prefix inet;
  }

  revision "2006-09-01" {
```

```
      description "Initial servers data model";
  }

  /*  A set of server structures  */
  container servers {
    list server {
      key name;
      max-elements 64;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ip-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
    }
  }
}
```

Since we are using CDB here, ConfD will keep an XML tree conforming to the above data model in its internal persistent XML database.

We start by saving the YANG module to a file, `servers.yang` and compile and link the data model into a single `servers.fxs` which is the binary format, used by ConfD, of a YANG module.

```
$ confdc -c servers.yang
```

We then proceed to use the `--emit-h` flag to generate a .h file which contains the namespace symbol (`servers__ns`) which we need in order to use the CDB API.

```
$ confdc --emit-h servers.h servers.fxs
$ head servers.h

#ifndef _SERVERS_H_
#define _SERVERS_H_

#ifndef servers__ns
#define servers__ns 686487091
#define servers__ns_id "http://example.com/ns/servers"
#define servers__ns_uri "http://example.com/ns/servers"
#endif
```

Once we have compiled the YANG module, we can start ConfD. We need to provide a configuration file to ConfD which indicates that we want ConfD to store the configuration.

The relevant parts from the `confd.conf` configuration file are:

```
<loadPath>
    <dir>/etc/confd</dir>
</loadPath>

...

<cdb>
    <enabled>true</enabled>
    <dbDir>/var/confd/cdb</dbDir>
</cdb>
```

The newly generated .fxs file must be copied to the directory `/etc/confd` and the directory `/var/confd/cdb` must exist and be writable. Thus:

```
$ cp servers.fxs /etc/confd
$ mkdir /var/confd/cdb
$ confd -v --foreground
```

By far the easiest way to populate the database with some actual data is to run the CLI.

```
$ confd_cli -u admin
admin connected from 127.0.0.1 using console on buzz
admin@buzz> configure private
Entering configuration mode "private"
admin@buzz% set servers server www
admin@buzz% set servers server www port 80
admin@buzz% set servers server www ip 192.168.128.1
admin@buzz% commit
Configuration committed
admin@buzz> show configuration
servers {
    server www {
        ip 192.168.128.1;
        port 80;
    }
}
```

Now the database is populated with a single server instance.

What remains to conclude our simple example is to write our application - our managed object - the code that uses the configuration data in the database. The implied meaning of the `servers.yang` YANG module is that the managed object would start and stop the services in the configuration. We will not do that; we will merely show how to read the configuration from the CDB database and react to changes in CDB.

The code is straightforward. We are using the API functions from `libconfd.so`. The CDB API is fully described in the UNIX man page confd_lib_cdb(3).

Main looks like this:

```
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_SILENT);

    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 confd_connect() to confd \n");

    if ((status =
```

```
         cdb_subscribe(subsock, 3, servers__ns, &spoint,"/servers"))
        != CONFD_OK) {
        fprintf(stderr, "Terminate: subscribe %d\n", status);
        exit(1);
    }
    if (cdb_subscribe_done(subsock) != CONFD_OK)
        confd_fatal("cdb_subscribe_done() failed");

    if ((status = read_conf(&addr)) != CONFD_OK) {
        fprintf(stderr, "Terminate: read_conf %d\n", status);
        exit(1);
    }

    /* ... */
```

The code initializes the library, reads the configuration and creates a socket to CDB. One socket is a read socket and it is used to read configuration data by means of CDB API read functions, while the other is a subscription socket. The subscription socket must be part of the client `poll()` set. Whenever data arrives on the subscription socket, the client invokes a CDB API function, `cdb_read_subscription_socket()` on the subscription socket. The subscription model will be explained further later in this chapter.

The `read_conf()` function reads the configuration data from CDB and stores it in local ephemeral (temporary) data structures. We have:

```
#include "servers.h"

struct server {
    char name[BUFSIZ];
    struct in_addr ip;
    unsigned int port;
};

static struct server running_db[64];
static int num_servers = 0;

static int read_conf(struct sockaddr_in *addr)
{
    int rsock, i, n, st = CONFD_OK;
    struct in_addr ip;
    u_int16_t port;
    char buf[BUFSIZ];

    if ((rsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        return CONFD_ERR;

    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, servers__ns);
    num_servers = 0;
    if ((n = cdb_num_instances(rsock, "/servers/server")) < 0) {
        cdb_close(rsock);
        return n;
    }
    num_servers = n;
    for(i=0; i<n; i++) {
```

```
        if ((st = cdb_get_str(rsock, buf, BUFSIZ,
                              "/servers/server[%d]/name",i)) != CONFD_OK)
            break;
        if ((st = cdb_get_ipv4(
                 rsock, &ip,"/servers/server[%d]/ip",i))!= CONFD_OK)
            break;
        if ((st = cdb_get_u_int16(
                 rsock, &port, "/servers/server[%d]/port",i)) != CONFD_OK)
            break;
        strcpy(running_db[i].name, buf);
        running_db[i].ip.s_addr = ip.s_addr;
        running_db[i].port = port;
    }
    cdb_close(rsock);
    return st;
}
```

The code first creates a read socket to ConfD by means of `cdb_connect()`. Following that, the code figures out how many `server` instances CDB has stored and then loops over all of those instances and reads the individual leaves with the different `cdb_get_` functions.

Finally we have our `poll()` loop. The subscription socket we created in `main()` must be added to the poll set - and whenever that file descriptor has IO ready to read we must act. When `subsock` is ready to read, the following code fragment should be executed:

```
        int sub_points[1];
        int reslen;

        if ((status = cdb_read_subscription_socket(subsock,
                                                   sub_points,
                                                   &reslen)) != CONFD_OK)
            exit(status);
        if (reslen > 0) {
            if ((status = read_conf(&addr)) != CONFD_OK)
                exit(1);
        }
        print_servers();        /* do something with data here... */
        if ((status =
             cdb_sync_subscription_socket(subsock, CDB_DONE_PRIORITY))
            != CONFD_OK) {
            exit(status);
        }
```

Instead of actually using the data we will merely print it to stdout when we receive any changes:

```
static void print_servers()
{
    int i;

    for (i=0; i < num_servers; i++) {
        printf("server %d: %s %s:%d\n", i, running_db[i].name,
               inet_ntoa(running_db[i].ip), running_db[i].port);
    }
}
```

5.4. Using keypaths

We'll go through all the CDB API functions used in the C code, but first a note on the path notation. Several of the API functions take a keypath as a parameter. A keypath leads down into the configuration data tree.

A keypath can be either absolute or relative. An absolute keypath starts from the root of the tree, while a relative path starts from the "current position" in the tree. They are differentiated by presence or absence of a leading "/". It is possible to change the "current position" with for example the `cdb_cd()` function.

XML elements that are containers for other XML elements, such as the `servers` container that contains multiple `server` instances, can be traversed using two different path notations. In our code above, we use the function `cdb_num_instances()` to figure out how many children a list has, and then traverse all children using a `[%d]` notation. The children of a list have an implicit numbering starting at 0. Thus the path: `/servers/server[2]/port` refers to the "port" leaf of the third `server` in the configuration. This numbering is only valid during the current CDB session. CDB is always locked for the duration of the read session.

We can also refer to list instances using the values of the keys of the list. Remember that we specified in the data model which leaf(s) in the XML structure were keys using the `key name` statement at the beginning of the list. In our case a `server` has the `name` leaf as key. So the path: `/servers/server{www}/ip` refers to the `ip` leaf of the server whose name is "www".

A YANG list may have more than one key. In the next section we will provide an example where we configure a DHCP daemon. That data model uses multiple keys and for example the path: `/dhcp/subNets/subNet{192.168.128.0 255.255.255.0}/routers` refers to the routers list of the subNet which has key `"192.168.128.0 255.255.255.0"`.

The syntax for keys is a space separated list of key values enclosed within curly brackets: `{ Key1 Key2 ...}`

Which version of bracket notation to use depends on the situation. For example the bracket notation is normally used when looping through all instances. As a convenience all functions expecting keypaths accept formatting characters and accompanying data items. For example `cdb_get("server[%d]/ifc{%s}/mtu", 2, "eth0")` to fetch the MTU of the third server instance's interface named "eth0". Using relative paths and `cdb_pushd()` it is possible to write code that can be re-used for common sub-trees. An example of this is presented further down.

The current position also includes the namespace. To read elements from a different namespace use the `cdb_set_namespace()` function.

5.5. A session

It is important to consider that CDB is locked for writing during a read session using the C API. A session starts with `cdb_start_session()` and the lock is not released until the `cdb_end_session()` (or the `cdb_close()`) call. CDB will also automatically release the lock if the socket is closed for some other reason, such as program termination.

In the example above we created a new socket each time we called `read_conf()`. It is also possible to re-use an existing connection.

Example 5.2. Pseudo code showing several sessions reusing one connection

```
cdb_connect(s);

..

cdb_start_session(s); /* Start session and take CDB lock */
  cdb_cd();
  cdb_get();
cdb_end_session(s);   /* lock is released */
```

```
..

cdb_start_session(s); /* Start session and take CDB lock */
  cdb_get();
cdb_end_session(s);   /* lock is released */

..

cdb_close(s);
```

5.6. CDB subscriptions

The CDB subscription mechanism allows an external program to be notified when different parts of the configuration changes. At the time of notification it is also possible to iterate through the changes written to CDB. Subscriptions are always towards the running datastore (it is not possible to subscribe to changes to the startup datastore). Subscriptions towards the operational data kept in CDB are also possible, but the mechanism is slightly different, see below.

The first thing to do is to inform CDB which paths we want to subscribe to, registering a path returns a subscription point identifier. This is done with the `cdb_subscribe()` function. Each subscriber can have multiple subscription points, and there can be many different subscribers. Every point is defined through a path - similar to the paths we use for read operations, with the exception that instead of fully instantiated paths to list instances we can selectively use tagpaths.

We can subscribe either to specific leaves, or entire subtrees. Explaining this by example we get:

`/named/options/pid-file`
: a subscription to a leaf. Only changes to this leaf will generate a notification.

`/servers`
: Means that we subscribe to any changes in the subtree rooted at `/servers`. This includes additions or removals of `server` instances, as well as changes to already existing `server` instances.

`/servers/server{www}/ip`
: Means that we only want to be notified when the server "www" changes its ip address.

`/servers/server/ip`
: Means we want to be notified when the leaf `ip` is changed in *any* server instance.

When adding a subscription point the client must also provide a priority, which is an integer. As CDB is changed, the change is part of a transaction. For example the transaction is initiated by a **commit** operation from the CLI or a **candidate-commit** operation in NETCONF resulting in the running database being modified. As the last part of the transaction CDB will generate notifications in lock-step priority order. First all subscribers at the lowest numbered priority are handled, once they all have replied and synchronized by calling `cdb_sync_subscription_socket()` the next set - at the next priority level - is handled by CDB. Not until all subscription points have been acknowledged is the transaction complete. This implies that if the initiator of the transaction was for example a **commit** command in the CLI, the command will hang until notifications have been acknowledged.

Note that even though the notifications are delivered within the transaction it is not possible for a subscriber to reject the changes (since this would break the two-phase commit protocol used by the ConfD backplane towards all data-providers).

When a client is done subscribing it needs to inform ConfD it is ready to receive notifications. This is done by first calling `cdb_subscribe_done()`, after which the subscription socket is ready to be polled.

As a subscriber has read its subscription notifications using `cdb_read_subscription_socket()` it can iterate through the changes that caused the particular subscription notification using the `cdb_diff_iterate()` function. It is also possible to start a new read-session to the CDB_PRE_COMMIT_RUNNING database to read the running database as it was before the pending transaction.

Subscriptions towards the operational data in CDB are similar to the above, but due to the fact that the operational data store is designed for light-weight access, and thus does not have transactions and normally avoids the use of any locks, there are several differences - in particular:

- Subscription notifications are only generated if the writer obtains a "subscription lock", by using the `cdb_start_session2()` function with the CDB_LOCK_REQUEST flag, see the confd_lib_cdb(3) manual page. It is possible to obtain a "subscription lock" for a subtree of the operational data store by using the CDB_LOCK_PARTIAL flag.

- Subscriptions are registered by using the `cdb_subscribe2()` function with type `CDB_SUB_OPERATIONAL` (or `cdb_oper_subscribe()`) rather than `cdb_subscribe()`.

- No priorities are used.

- Neither the writer that generated the subscription notifications nor other writes to the same data are blocked while notifications are being delivered. However the subscription lock remains in effect until notification delivery is complete.

- The previous value for a modified leaf is not available when using the `cdb_diff_iterate()` function.

Essentially a write operation towards the operational data store, combined with the subscription lock, takes on the role of a transaction for configuration data as far as subscription notifications are concerned. This means that if operational data updates are done with many single-element write operations, this can potentially result in a lot of subscription notifications. Thus it is a good idea to use the multi-element `cdb_set_object()` etc functions for updating operational data that applications subscribe to.

Since write operations that do not attempt to obtain a subscription lock are allowed to proceed even during notification delivery, it is the responsibility of the applications using the operational data store to obtain the lock as needed when writing. E.g. if subscribers should be able to reliably read the exact data that resulted from the write that triggered their subscription, a subscription lock must always be obtained when writing that particular set of data elements. One possibility is of course to obtain a lock for *all* writes to operational data, but this may have an unacceptable performance impact.

To view registered subscribers use the **confd --status** command. For details on how to use the different subscription functions see the confd_lib_cdb(3) manual page.

5.7. Reconnect

If ConfD is restarted, our CDB sockets obviously die. The correct thing to do then is to re-open the cdb sockets and re-read the configuration. In the case of a high availability setup this also applies. If we are connected to one ConfD node and that node dies, we must reconnect to another ConfD node and read/subscribe to the configuration from that node.

If the configuration has not changed we do not want to restart our managed objects, we just want to reconnect our CDB sockets. The API function `cdb_get_txid()` will read the last transaction id from our cdb socket. The id is guaranteed to be unique. We issue the call `cdb_get_txid()` on the data socket and we must not have an active read session on that socket while issuing the call.

Example 5.3. Pseudo code demonstrating how to avoid re-reading the configuration

```
 struct cdb_txid prev_stamp;

 cdb_connect(s);
 load_config(s);
 cdb_get_txid(s, &prev_stamp);
...
 subscribe(....);
 while (1) {
    poll(...);
    if (has_new_data(s)) {
        load_config(s);
        cdb_get_txid(s, &prev_stamp);
    }
    else if (is_closed(s)) {
         struct cdb_txid new_stamp;
         cdb_connect(s);
         cdb_get_txid(s, &new_stamp);
         if ((prev_stamp.s1 == new_stamp.s1) &&
             (prev_stamp.s2 == new_stamp.s2) &&
             (prev_stamp.s3 == new_stamp.s3) &&
             (strcmp(prev_stamp.master, new_stamp.master) == 0)) {
             /* no need to re-read config, it hasn't changed */
             continue;
         }
         else {
             load_config(s);
             cdb_get_txid(s, &prev_stamp);
        }
    }
 }
```

5.8. Loading initial data into CDB

When ConfD starts for the first time, assuming CDB is enabled, the CDB database is empty. CDB is configured to store its data in a directory as in:

```
<cdb>
    <enabled>true</enabled>
    <dbDir>/var/confd/cdb</dbDir>
</cdb>
```

At startup, when CDB is empty, i.e. no database files are found in the CDB directory, CDB will try to initialize the database from all instantiated XML documents found in the CDB directory. This is the mechanism we use to have an empty database initialized to some default setup.

This feature can be used to for example reset the configuration back to some factory setting or some such.

For example, assume we have the data model from Example 5.1, “a simple server data model, `servers.yang`”. Furthermore, assume CDB is empty, i.e. no database files at all reside under `/var/confd/cdb`. However we do have a file, `/var/confd/cdb/foobar.xml` containing the following data:

```
<servers:servers xmlns:servers="http://example.com/ns/servers">
  <servers:server>
    <servers:name>www</servers:name>
    <servers:ip>192.168.3.4</servers:ip>
    <servers:port>88</servers:port>
```

```
  </servers:server>
  <servers:server>
    <servers:name>www2</servers:name>
    <servers:ip>192.168.3.5</servers:ip>
    <servers:port>80</servers:port>
  </servers:server>
  <servers:server>
    <servers:name>smtp</servers:name>
    <servers:ip>192.168.3.4</servers:ip>
    <servers:port>25</servers:port>
  </servers:server>
  <servers:server>
    <servers:name>dns</servers:name>
    <servers:ip>192.168.3.5</servers:ip>
    <servers:port>53</servers:port>
  </servers:server>
</servers:servers>
```

CDB will be initialized from the above XML document. The feature of initializing CDB with some predefined set of XML elements is used to initialize the AAA database. This is described in Chapter 14, *The AAA infrastructure*.

All files ending in `.xml` will be loaded (in an undefined order) and committed in a single transaction when CDB enters start phase 1 (see Section 27.5, “Starting ConfD” for more details on start phases). The format of the init files is rather lax in that it is not required that a complete instance document following the data-model is present, much like the NETCONF `edit-config` operation. It is also possible to wrap multiple top-level tags in the file with a surrounding config tag, like this:

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  ...
</config>
```

5.9. Automatic schema upgrades and downgrades

Software upgrades and downgrades represent one of the main problems of managing configuration data of network devices. Each software release for a network device is typically associated with a certain version of configuration data layout, i.e. a schema. In ConfD the schema is the data model stored in the .fxs files. Once CDB has initialized it also stores a copy of the schema associated with the data it holds.

Every time ConfD starts, CDB will check the current contents of the .fxs files with its own copy of the schema files. If CDB detects any changes in the schema it initiates an upgrade transaction. In the simplest case CDB automatically resolves the changes and commits the new data before ConfD reaches start-phase one.

An example of how the CDB automatically handles upgrades follows. For version 1.0 of the "forest configurator" software project the following YANG module is used:

Example 5.4. Version 1.0 of the forest module

```
module forest {
  namespace "http://example.com/ns/forest";
  prefix forest;

  revision "2006-09-01" {
      description "Initial forest model";
```

```
  }

  container forest {
    list tree {
      key name;
      min-elements 2;
      max-elements 1024;
      leaf name {
        type string;
      }
      leaf height {
        type uint8;
        mandatory true;
      }
      leaf type {
        type string;
        mandatory true;
      }
    }
    list flower {
      key name;
      max-elements 1024;
      leaf name {
        type string;
      }
      leaf type {
        type string;
        mandatory true;
      }
      leaf color {
        type string;
        mandatory true;
      }
    }
  }
}
```

The YANG module will be mounted at / in the larger compounded data model tree. We start ConfD and populate the CDB, e.g. by using the ConfD CLI. The programmer then writes C code which reads these items from the configuration database.

To demonstrate the automatic upgrade, we will assume that CDB is populated with the instance data in Example 5.5, “Initial forest instance document”

Example 5.5. Initial forest instance document

```
<forest xmlns="http://example.com/ns/forest">
  <tree>
    <name>George</name><height>10</height><type>oak</type>
  </tree>
  <tree>
    <name>Eliza</name><height>15</height><type>oak</type>
  </tree>
  <tree>
    <name>Henry</name><height>12</height><type>pine</type>
  </tree>
  <flower>
    <name>Sebastian</name><type>dandelion</type><color>yellow</color>
  </flower>
```

```
  <flower>
    <name>Alvin</name><type>tulip</type><color>white</color>
  </flower>
</forest>
```

During the development of the next version of the "forest configurator" software project a couple of changes were made to the configuration data schema. The `tree` list height was found to need more than 256 possible values and expanded to a 32-bit integer, and two new leaves `color` and `birthday` were added. The list `flower` had an optional `edible` leaf added, and the `color` changed type to a more strict enumeration type. The result is in Example 5.6, “Version 2.0 of the forest module”.

Example 5.6. Version 2.0 of the forest module

```
module forest {
  namespace "http://example.com/ns/forest";
  prefix forest;

  import tailf-xsd-types {
    prefix xs;
  }

  revision "2009-10-01" {
      description
          "Needed room for taller trees.
           Flowers can be edible.";
  }

  revision "2008-09-01" {
      description "Initial forest model";
  }

  typedef colorType {
    type enumeration {
      enum unknown;
      enum blue;
      enum yellow;
      enum red;
      enum green;
    }
  }
  container forest {
    list tree {
      key name;
      min-elements 2;
      max-elements 1024;
      leaf name {
        type string;
      }
      leaf height {
        type int32;
        mandatory true;
      }
      leaf birthday {
        type xs:date;
        default 2006-09-01;
      }
      leaf color {
        type colorType;
        default unknown;
```

```
      }
      leaf type {
        type string;
        mandatory true;
      }
    }
    list flower {
      key name;
      max-elements 1024;
      leaf name {
        type string;
      }
      leaf type {
        type string;
        mandatory true;
      }
      leaf edible {
        type empty;
      }
      leaf color {
        type colorType;
        default unknown;
      }
    }
  }
}
```

After compiling this new version of the YANG module into an fxs file using **confdc** and restarting ConfD with the new schema, CDB automatically detects that the namespace `http://example.com/ns/forest` has been modified. CDB will then update the schema *and* the contents of the database. When ConfD has started the data in the database now looks like in Example 5.7, “Forest instance document after upgrade”

Example 5.7. Forest instance document after upgrade

```
    <forest xmlns="http://example.com/ns/forest">
      <tree>
        <name>Eliza</name>
        <height>15</height>
        <birthday>2006-09-01</birthday>
        <color>unknown</color>
        <type>oak</type>
      </tree>
      <tree>
        <name>George</name>
        <height>10</height>
        <birthday>2006-09-01</birthday>
        <color>unknown</color>
        <type>oak</type>
      </tree>
      <tree>
        <name>Henry</name>
        <height>12</height>
        <birthday>2006-09-01</birthday>
        <color>unknown</color>
        <type>pine</type>
      </tree>
      <flower>
        <name>Alvin</name>
```

```
        <type>tulip</type>
        <color>unknown</color>
      </flower>
      <flower>
        <name>Sebastian</name>
        <type>dandelion</type>
        <color>yellow</color>
      </flower>
    </forest>
```

Let's follow what CDB does by checking the devel log. The devel log is meant to be used as support while the application is developed. It is enabled in `confd.conf` as shown in Example 5.8, “Enabling the developer log”.

Example 5.8. Enabling the developer log

```
<developerLog>
  <enabled>true</enabled>
  <file>
    <enabled>true</enabled>
    <name>/var/confd/log/devel.log</name>
  </file>
</developerLog>
<developerLogLevel>trace</developerLogLevel>
```

Example 5.9. Developer log entries resulting from upgrade

```
upgrade: http://example.com/ns/forest -> http://example.com/ns/forest
upgrade: /forest/flower/{"Alvin"}/color: -> unknown (default because old value white does n
upgrade: /forest/flower/{"Sebastian"}/color: -> yellow (but with new type)
upgrade: /forest/tree/{"Eliza"}/birthday: added, with default (2006-09-01)
upgrade: /forest/tree/{"Eliza"}/color: added, with default (unknown)
upgrade: /forest/tree/{"Eliza"}/height: -> 15 (but with new type)
upgrade: /forest/tree/{"George"}/birthday: added, with default (2006-09-01)
upgrade: /forest/tree/{"George"}/color: added, with default (unknown)
upgrade: /forest/tree/{"George"}/height: -> 10 (but with new type)
upgrade: /forest/tree/{"Henry"}/birthday: added, with default (2006-09-01)
upgrade: /forest/tree/{"Henry"}/color: added, with default (unknown)
upgrade: /forest/tree/{"Henry"}/height: -> 12 (but with new type)
```

CDB can automatically handle the following changes to the schema:

Deleted elements
: When an element is deleted from the schema, CDB simply deletes it (and any children) from the database.

Added elements
: If a new element is added to the schema it needs to either be optional, dynamic, or have a default value. New elements with a default are added set to their default value. New dynamic or optional elements are simply noted as a schema change.

Re-ordering elements
: An element with the same name, but in a different position on the same level, is considered to be the same element. If its type hasn't changed it will retain its value, but if the type has changed it will be upgraded as described below.

Type changes
: If a leaf is still present but its type has changed, automatic coercions are performed, so for example integers may be transformed to their string representation if the type changed from e.g. int32 to string.

Automatic type conversion succeeds as long as the string representation of the current value can be parsed into its new type. (Which of course also implies that a change from a smaller integer type, e.g. int8, to a larger type, e.g. int32, succeeds for any value - while the opposite will not hold, but might!)

If the coercion fails, any supplied default value will be used. If no default value is present in the new schema the *automatic* upgrade will fail.

Type changes when user-defined types are used are also handled automatically, provided that some straightforward rules are followed for the type definitions. Read more about user-defined types in the confd_types(3) manual page, which also describes these rules.

Hash changes
: When a hash value of particular element has changed (due to an addition of, or a change to, a `tailf:id-value` statement) CDB will update that element.

Key changes
: When a key of a list is modified, CDB tries to upgrade the key using the same rules as explained above for adding, deleting, re-ordering, change of type, and change of hash value. If automatic upgrade of a key fails the entire list instance will be deleted.

Default values
: If a leaf has a default value, which has not been changed from its default, then the automatic upgrade will use the new default value (if any). If the leaf value has been changed from the old default, then that value will be kept.

Adding / Removing namespaces
: If a namespace no longer is present after an upgrade, CDB removes all data in that namespace. When CDB detects a new namespace, it is initialized with default values.

Changing to/from operational
: Elements that previously had `config false` set that are changed into database elements will be treated as a added elements. In the opposite case, where data elements in the new data model are tagged with `config false`, the elements will be deleted from the database.

Callpoint changes
: CDB only considers the part of the data model in YANG modules that do not have external callpoints (see Chapter 7, *The external database API*). But while upgrading, CDB does handle moving subtrees into CDB from a callpoint and vice versa. CDB simply considers these as added and deleted schema elements.

  Thus an application can be developed using CDB in the first development cycle. When the external database component is ready it can easily replace CDB without changing the schema.

Should the *automatic* upgrade fail, exit codes and log-entries will indicate the reason (see Section 27.11, “Disaster management”).

5.10. Using initialization files for upgrade

As described earlier, when ConfD starts with an empty CDB database, CDB will load all instantiated XML documents found in the CDB directory and use these to initialize the the database. We can also use this mechanism for CDB upgrade, since CDB will again look for files in the CDB directory ending in `.xml` when doing an upgrade.

This allows for handling many of the cases that the automatic upgrade can not do by itself, e.g. addition of mandatory leaves (without default statements), or multiple instances of new dynamic containers. Most

of the time we can probably simply use the XML init file that is appropriate for a fresh install of the new version also for the upgrade from a previous version.

When using XML files for initialization of CDB, the complete contents of the files is used. On upgrade however, doing this could lead to modification of the user's existing configuration - e.g. we could end up resetting data that the user has modified since CDB was first initialized. For this reason two restrictions are applied when loading the XML files on upgrade:

- Only data for elements that are new as of the upgrade (i.e. elements that did not exist in the previous schema) will be considered.
- The data will only be loaded if all old (i.e. previously existing) optional/dynamic parent elements and instances exist in the current configuration.

To clarify this, we will look again at Example 5.1, “a simple server data model, `servers.yang`”. In version 1.5 of the server manager, it was realized that the data model had a serious shortcoming: There was no way to specify the protocol to use, TCP or UDP. To fix this, another leaf was added to the `/servers/server` list, and the new YANG module looks like this:

Example 5.10. Version 1.5 of the servers.yang module

```
module servers {
  namespace "http://example.com/ns/servers";
  prefix servers;

  import ietf-inet-types {
    prefix inet;
  }

  revision "2007-06-01" {
      description "added protocol.";
  }

  revision "2006-09-01" {
      description "Initial servers data model";
  }

  /*  A set of server structures  */
  container servers {
    list server {
      key name;
      max-elements 64;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ip-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
      leaf protocol {
        type enumeration {
            enum tcp;
            enum udp;
        }
        mandatory true;
```

```
      }
    }
  }
}
```

Since it was considered important that the user explicitly specified the protocol, the new leaf was made mandatory. Of course the XML init file was updated to include this leaf, and now looks like this:

```
<servers:servers xmlns:servers="http://example.com/ns/servers">
  <servers:server>
    <servers:name>www</servers:name>
    <servers:ip>192.168.3.4</servers:ip>
    <servers:port>88</servers:port>
    <servers:protocol>tcp</servers:protocol>
  </servers:server>
  <servers:server>
    <servers:name>www2</servers:name>
    <servers:ip>192.168.3.5</servers:ip>
    <servers:port>80</servers:port>
    <servers:protocol>tcp</servers:protocol>
  </servers:server>
  <servers:server>
    <servers:name>smtp</servers:name>
    <servers:ip>192.168.3.4</servers:ip>
    <servers:port>25</servers:port>
    <servers:protocol>tcp</servers:protocol>
  </servers:server>
  <servers:server>
    <servers:name>dns</servers:name>
    <servers:ip>192.168.3.5</servers:ip>
    <servers:port>53</servers:port>
    <servers:protocol>udp</servers:protocol>
  </servers:server>
</servers:servers>
```

We can then just use this new init file for the upgrade, and the existing server instances in the user's configuration will get the new `/servers/server/protocol` leaf filled in as expected. However some users may have deleted some of the original servers from their configuration, and in those cases we obviously do not want those servers to get re-created during the upgrade just because they are present in the XML file - the above restrictions make sure that this does not happen. Here is what the configuration looks like after upgrade if the "smtp" server has been deleted before upgrade:

```
    <servers xmlns="http://example.com/ns/servers">
      <server>
        <name>dns</name>
        <ip>192.168.3.5</ip>
        <port>53</port>
        <protocol>udp</protocol>
      </server>
      <server>
        <name>www</name>
        <ip>192.168.3.4</ip>
        <port>88</port>
        <protocol>tcp</protocol>
      </server>
      <server>
        <name>www2</name>
        <ip>192.168.3.5</ip>
        <port>80</port>
        <protocol>tcp</protocol>
```

```
      </server>
    </servers>
```

This example also implicitly shows a limitation with this method: If the user has created additional servers, the new XML file will not specify what protocol to use for those servers, and the upgrade cannot succeed unless the external program method is used, see below. However the example is a bit contrived - in practice this limitation is rarely a problem: It does not occur for new lists or optional elements, nor for new mandatory elements that are not children of old lists. And in fact correctly adding this "protocol" leaf for user-created servers would require user input - it can not be done by *any* fully automated procedure.

Note

Since CDB will attempt to load all `*.xml` files in the CDB directory at the time of upgrade, it is important to not leave XML init files from a previous version that are no longer valid there.

It is always possible to write an external program to change the data before the upgrade transaction is committed. This will be explained in the following sections.

5.11. Using MAAPI to modify CDB during upgrade

To take full control over the upgrade transaction, ConfD must be started using the `--start-phase{0,1,2}` command line options. When ConfD is started using the `--start-phase0` option CDB will initiate, and if it detects an upgrade situation the upgrade transaction will be created, and all *automatic* upgrades will be performed. After which ConfD simply waits, either for a MAAPI connection or **confd --start-phase1**.

Whenever changes to the schema cannot be handled automatically, or when the application programmer wants more control over how the data in the upgraded database is populated it is possible to use MAAPI to attach and write to the upgrade transaction in progress (see Chapter 22, *The Management Agent API* for details on this API).

Using the `maapi_attach_init()` function call an external program can attach to the upgrade transaction during `phase0`. For example a program that creates the optional container `edible` on each `flower` in the previous forest example would look like this:

Example 5.11. Writing to an upgrade transaction using MAAPI

```
int th;

maapi_attach_init(ms, &th);
maapi_set_namespace(ms, th, simple__ns);

maapi_init_cursor(sock, th, &mc, "/forest/flower");
maapi_get_next(&mc);
while (mc.n != 0) {
    maapi_create(sock, th, "/forest/flower{%x}/edible", &mc.keys[0]);
    maapi_get_next(&mc);
}
maapi_destroy_cursor(&mc);

exit(0);
```

Note the use of the special `maapi_attach_init()` function, it attaches the MAAPI socket to the upgrade transaction (or init transaction) and returns (through the second argument) the transaction handle

we need to make further MAAPI calls. This special upgrade transaction is only available during phase0. Once we call **confd --start-phase1** the transaction will be committed.

This method can also be combined with the init file usage described in the previous section - the data from the init file will be applied immediately following the automatic conversions at the beginning of `phase0`, and the external program can then use MAAPI to modify or complement the result.

5.12. More complex schema upgrades

In the previous section we showed how to use MAAPI to access the upgrade transaction *after* the automatic upgrade had taken place. But this means that CDB has deleted all values that are no longer part of the schema. Well, not quite yet. In start-phase0 it is possible to use all the CDB C-API calls to access the data using the schema from the database as it looked *before* the automatic upgrade. That is, the *complete* database as it stood before the upgrade is still available to the application. This allows us to write programs that transfer data in an application specific way between software releases.

Say, for example, that the developers of the Example 5.1, “a simple server data model, `servers.yang`” now has decided that having all servers under one top-element is not enough for their 2.0 release. They want to instead have different categories of servers under different server types. Also a new IP address is added to each server, the `adminIP`. The new version of `servers.yang` could then look like this (version 1.5 has not been merged to the 2.0 branch yet):

Example 5.12. Version 2 of the servers.yang module

```
module servers {

  namespace "http://example.com/ns/servers";
  prefix servers;

  import ietf-inet-types {
    prefix inet;
  }

  revision "2007-07-15" {
      description "Split servers into www and others";
  }

  revision "2006-09-01" {
      description "Initial servers data model";
  }

  container servers {
    list www {
      key name;
      max-elements 32;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ip-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
      leaf adminIP {
```

```
        type inet:ip-address;
        mandatory true;
      }
    }
    list others {
      key name;
      max-elements 32;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ip-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
      leaf adminIP {
        type inet:ip-address;
        mandatory true;
      }
    }
  }
}
```

The plan for upgrading users with the old version of the software is to transfer all servers with their name containing the letters "www" or whose port is equal to 80, to the `/servers/www` subtree, and all the others to the `/servers/others` subtree. The new `adminIP` element will be initialized to 10.0.0.1 for the first server, and then increased by one for each server found in the old database. The code to perform this operation would have to be written like this:

Example 5.13. The `upgrade()` function of `server_upgrade.c`

```
static struct sockaddr_in addr; /* Keeps address to confd daemon */

/* confd_init()  must be called before calling this function */
/* ms and cs are assumed to be valid sockets */
static void upgrade(int ms, int cs)
{
    int th;
    int i, n;
    static struct in_addr admin_ip;

    cdb_connect(cs, CDB_READ_SOCKET,
                (struct sockaddr *)&addr, sizeof(addr));
    cdb_start_session(cs, CDB_RUNNING);
    cdb_set_namespace(cs, servers__ns);

    maapi_connect(ms, (struct sockaddr *)&addr, sizeof(addr));
    maapi_attach_init(ms, &th);
    maapi_set_namespace(ms, th, servers__ns);

    admin_ip.s_addr = htonl(0x0a000001); /* initialize to 10.0.0.1 */
    n = cdb_num_instances(cs, "/servers/server");
    printf("servers = %d\n", n);
    for (i=0; i < n; i++) {
        char name[128];
        confd_value_t ip, port, aip;
```

```
        char *dst;

        /* read old database using cdb_* API */
        cdb_get_str(cs, name, 128, "/servers/server[%d]/name", i);
        cdb_get(cs, &ip,   "/servers/server[%d]/ip", i);
        cdb_get(cs, &port, "/servers/server[%d]/port", i);

        if ((CONFD_GET_UINT16(&port) == 80) ||
            (strstr(name, "www") != NULL)) {
            dst = "/servers/www{%s}";
        } else {
            dst = "/servers/others{%s}";
        }
        /* now create entries in the new database using maapi */
        maapi_create(ms, th, dst, name);
        maapi_pushd(ms, th, dst, name);
        maapi_set_elem(ms, th, &ip, "ip");
        maapi_set_elem(ms, th, &port, "port");
        CONFD_SET_IPV4(&aip, admin_ip);
        maapi_set_elem(ms, th, &aip, "adminIP");
        admin_ip.s_addr = htonl(ntohl(admin_ip.s_addr) + 1);
        maapi_popd(ms, th);
    }

    cdb_end_session(cs);
    cdb_close(cs);
}
```

It would of course be wise to add error checks to the code above. Also note that choosing new values for added elements can be done in a number of different ways, perhaps the end-user needs to be prompted, perhaps the data resides elsewhere on the device.

Now assuming the data in CDB is as in the "Initialization data for CDB" figure in the section above, then running the upgrade code would be done in the following order.

```
$ confd --stop
          ... Install new versions of software and fxs files ...
$ confd --start-phase0
$ server_upgrade
$ confd --start-phase1
          ... Start other internal daemons ...
$ confd --start-phase2
```

Finally, after ConfD is up and running after the upgrade the data would look like this.

```
    <servers xmlns="http://example.com/ns/servers">
      <www>
        <name>www</name>
        <ip>192.168.3.4</ip>
        <port>88</port>
        <adminIP>10.0.0.3</adminIP>
      </www>
      <www>
        <name>www2</name>
        <ip>192.168.3.5</ip>
        <port>80</port>
        <adminIP>10.0.0.4</adminIP>
      </www>
      <others>
        <name>dns</name>
```

```
      <ip>192.168.3.5</ip>
      <port>53</port>
      <adminIP>10.0.0.1</adminIP>
    </others>
    <others>
      <name>smtp</name>
      <ip>192.168.3.4</ip>
      <port>25</port>
      <adminIP>10.0.0.2</adminIP>
    </others>
  </servers>
```

ConfD does not impose any specific meaning to "version" - any change in the data model is an upgrade situation as far as CDB is concerned. ConfD also does not force the application programmer to handle software releases in a specific way, as each application may have very different needs and requirements in terms of footprint and storage etc.

5.13. The full dhcpd example

As an example, we show how to integrate dhcpd - the ISC DHCP daemon - under ConfD.

Assume we have Example 5.14, “A YANG module describing a dhcpd server configuration” in a file called `dhcpd.yang`.

Example 5.14. A YANG module describing a dhcpd server configuration

```
module dhcpd {
  namespace "http://example.com/ns/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;
    }
  }

  grouping subNetworkType {
    leaf net {
      type inet:ip-address;
    }
    leaf mask {
      type inet:ip-address;
    }
    container range {
      presence "";
      leaf dynamicBootP {
        type boolean;
        default false;
      }
```

```
      leaf lowAddr {
        type inet:ip-address;
        mandatory true;
      }
      leaf hiAddr {
        type inet:ip-address;
      }
    }
    leaf routers {
      type string;
    }
    leaf maxLeaseTime {
      type xs:duration;
      default PT7200S;
    }
  }
  container dhcp {
    leaf defaultLeaseTime {
      type xs:duration;
      default PT600S;
    }
    leaf maxLeaseTime {
      type xs:duration;
      default PT7200S;
    }
    leaf logFacility {
      type loglevel;
      default local7;
    }
    container SubNets {
      container subNet {
        uses subNetworkType;
      }
    }
    container SharedNetworks {
      list sharedNetwork {
        key name;
        max-elements 1024;
        leaf name {
          type string;
        }
        container SubNets {
          container subNet {
            uses subNetworkType;
          }
        }
      }
    }
  }
}
```

We use some interesting constructs in this module. We define a grouping and reuse it twice in the module. This is what we do when we want to reuse a data model structure defined somewhere else in the specification.

Now we have a data model which is fully usable with ConfD. Consider an actual `dhcpd.conf` file which looks like:

```
defaultleasetime 600;
maxleasetime 7200;
```

```
subnet 192.168.128.0 netmask 255.255.255.0 {
    range 192.168.128.60 192.168.128.98;
}
sharednetwork 22429 {
  subnet 10.17.224.0 netmask 255.255.255.0 {
     option routers rtr224.example.org;
  }
  subnet 10.0.29.0 netmask 255.255.255.0 {
     option routers rtr29.example.org;
  }
}
```

The above dhcp configuration would be represented by an XML structure that looks like this:

```
<dhcp>
  <maxLeaseTime>7200</maxLeaseTime>
  <defaultLeaseTime>600</defaultLeaseTime>
  <subNets>
    <subNet>
      <net>192.168.128.0</net>
      <mask>255.255.255.0</mask>
      <range>
        <lowAddr>192.168.128.60</lowAddr>
        <highAddr>192.168.128.98</highAddr>
      </range>
    </subNet>
  </subNets>
  <sharedNetworks>
    <sharedNetwork>
      <name>22429</name>
      <subNets>
        <subNet>
          <net>10.17.224.0</net>
          <mask>255.255.255.0</mask>
          <routers>rtr224.example.org</routers>
        </subNet>
        <subNet>
          <net>10.0.29.0</net>
          <mask>255.255.255.0</mask>
          <routers>rtr29.example.org</routers>
        </subNet>
      </subNets>
    </sharedNetwork>
  </sharedNetworks>
</dhcp>
```

The dhcp server subscribes to configuration changes and reconfigures when notified. For our purposes we write a small program which:

1. Reads the configuration database.

2. Writes `/etc/dhcp/dhcpd.conf` and HUPs the daemon.

The main function looks exactly like the example where we read the servers database with the exception that we establish a subscription socket for the path `/dhcp` instead of `/servers`. Whenever *any* configuration change occurs for anything related to dhcp, the program rereads the configuration from CDB, regenerates the `dhcpd.conf` file and HUPs the dhcp daemon.

Reading the dhcp configuration from CDB is done as follows:

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

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

    cdb_get_enum_value(rsock, &tmp, "/dhcp/logFacility");
    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:
        fprintf(fp, "log-facility local7\n");
        break;
    }
    n = cdb_num_instances(rsock, "/dhcp/subNets/subNet");
    for (i=0; i<n; i++) {
        cdb_cd(rsock, "/dhcp/subNets/subNet[%d]", i);
        do_subnet(rsock, fp);
    }
    n = cdb_num_instances(rsock, "/dhcp/SharedNetworks/sharedNetwork");
    for (i=0; i<n; i++) {
        unsigned char *buf;
        int buflen;
        int j, m;

        cdb_get_buf(rsock, &buf, &buflen,
                    "/dhcp/SharedNetworks/sharedNetwork[%d]/name");
        fprintf(fp, "shared-network %.*s {\n", buflen, buf);
        m = cdb_num_instances(
            rsock,
            "/dhcp/SharedNetworks/sharedNetwork[%d]/subNets/subNet", i);
        for (j=0; j<m; j++) {
            cdb_pushd(rsock, "/dhcp/SharedNetworks/sharedNetwork[%d]/"
                      "subNets/subNet[%d]", i, j);
            do_subnet(rsock, fp);
            cdb_popd(rsock);
        }
```

```
        fprintf(fp, "}\n");
    }
    fclose(fp);
    return cdb_close(rsock);
}
```

The code first establishes a read socket to CDB. Following that the code utilizes various CDB read functions to read the data. Look for example at how we extract the value `/dhcp/defaultLeaseTime` from CDB. Looking at Example 5.14, “A YANG module describing a dhcpd server configuration”, we see that the `type` of the leaf is xs:duration. There exists a special type safe version of `cdb_get()` which reads a duration value from the database, which we use as follows:

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

Alternatively, we could have written:

```
confd_value_t v;
struct confd_duration dur;

cdb_get(rsock, &v, "/dhcp/defaultLeaseTime");
dur = CONFD_GET_DURATION(&v)
fprintf(fp, "default-lease-time %d\n", duration_to_secs(&dur));
```

We figure out how many shared networks instances there are through the call to `cdb_num_instances()` and then refer to the individual instances through the string syntax `/dhcp/sharedNetworks/sharedNetwork[%d]`. The key of an individual shared network is, according to the data model, the `name` element but we do not care about that here, we iterate through each shared network instance using integers. These integers which obviously refer to shared network instances are only valid within this CDB session. Using the normal ConfD key syntax we can also refer to individual shared networks instances, e.g. `/dhcp/sharedNetworks/sharedNetwork{24-29}`. When we just wish to loop through a set of XML structures it is usually easier to use the `[%d]` key syntax.

Also note the call to `cdb_get_enum_value(rsock, &tmp, "/dhcp/logFacility");` which reads an enumeration. The `logFacility` element was defined as an enumerated type. Enumerations are represented as integers, both in our C code, but more importantly they are also represented as integers when we store the string in the CDB database on disk. I.e., we will not store the "logFacility" string "local7" over and over again on disk. Similarly in our C code we also get a (switchable) integer value as returned from `cdb_get_enum_value()`.

Especially interesting in the `read_conf()` code above is how we use `cdb_pushd()` in combination with `cdb_popd()`. The `cdb_pushd()` functions works like `cdb_cd()`, i.e. it changes the position in the data tree, with the difference that we can call `cdb_popd()` and return to where we were earlier in the XML tree. We traverse the `SharedNetworks` and execute

```
cdb_pushd(rsock, "/dhcp/SharedNetworks/sharedNetwork[%d]/"
                     "subNets/subNet[%d]", i, j);
do_subnet(rsock, fp);
cdb_popd(rsock);
```

Where the function `do_subnet()` reads in a subnet structure from CDB. The `do_subnet()` code works on relative paths as opposed to absolute paths.

The purpose of the example is to show how we can reuse the function `do_subnet()` and call it at different points in the XML tree. The function itself does not know whether it is reading a subnet under `/dhcp/subNets` or under `/dhcp/SharedNetworks/sharedNetwork/subNets`. Also noteworthy is that fact that utilizing pushd/popd makes for more efficient code since the part of the path leading up to the

pushed element is already parsed and verified to be correct. Thus the parsing and path verification does not have to be executed for each and every element.

```
static void do_subnet(int rsock, FILE *fp)
{
   struct in_addr ip;
   char buf[BUFSIZ];
   struct confd_duration dur;
   char *ptr;

   cdb_get_ipv4(rsock, &ip, "net");
   fprintf(fp, "subnet %s ", inet_ntoa(ip));
   cdb_get_ipv4(rsock, &ip, "mask");
   fprintf(fp, "netmask %s {\n", inet_ntoa(ip));

   if (cdb_exists(rsock, "range") == 1) {
       int bool;
       fprintf(fp, " range ");
       cdb_get_bool(rsock, &bool, "range/dynamicBootP");
       if (bool) fprintf(fp, " dynamic-bootp ");
       cdb_get_ipv4(rsock, &ip, "range/lowAddr");
       fprintf(fp, " %s ", inet_ntoa(ip));
       cdb_get_ipv4(rsock, &ip, "range/hiAddr");
       fprintf(fp, " %s ", inet_ntoa(ip));
       fprintf(fp, "\n");
   }
   cdb_get_str(rsock, &buf[0], BUFSIZ, "routers");

   /* replace space with comma */
   for (ptr = buf; ptr != '\0'; ptr++) {
       if (*ptr == ' ')
           *ptr = ',';
   }
   fprintf(fp, "routers %s\n", buf);
   cdb_get_duration(rsock, &dur, "maxLeaseTime");
   fprintf(fp, "max-lease-time %d\n", duration_to_secs(&dur));
}
```

Finally, we define the code fragment that must be executed by our application when there is IO ready to read on the subscription socket:

```
if (set[1].revents & POLLIN) {
   int sub_points[1];
   int reslen;

   if ((status = cdb_read_subscription_socket(subsock,
                                              &sub_points[0],
                                              &reslen)) != CONFD_OK) {
        fprintf(stderr, "terminate sub_read: %d\n", status);
        exit(1);
   }
   if (reslen > 0) {
        if ((status = read_conf(&addr)) != CONFD_OK) {
            fprintf(stderr, "Terminate: read_conf %d\n", status);
            exit(1);
        }
   }
   rename("dhcpd.conf.tmp", "/etc/dhcpd.conf");
   system("killall -HUP dhcpd");
   if ((status = cdb_sync_subscription_socket(subsock,
```

```
                                                CDB_DONE_PRIORITY))
        != CONFD_OK) {
        fprintf(stderr, "failed to sync subscription: %d\n", status);
        exit(1);
    }
}
```

Chapter 6. Operational Data

6.1. Introduction to Operational Data

In Chapter 3, *The YANG Data Modeling Language* we showed how to define data models in YANG. In Chapter 5, *CDB - The ConfD XML Database* we showed how to use CDB and also how to interface CDB to external daemons. In this chapter, we show how to write instrumentation code for read-only operational and statistics data.

Operational data is typically not kept in a database but read at runtime by instrumentation functions. This would for example be statistics counters contained inside the managed objects themselves. In this chapter we will show how to write such instrumentation functions in C.

An alternative approach to runtime operational data is to store the operational in CDB using a write interface. This will be described in Section 6.8, “Operational data in CDB”.

The configuration of the network device is modeled by a YANG module. This describes the data model of the device. We also need to write YANG modules for our operational data.

In the YANG data model, there can be restrictions on valid operational data. For example, a list might have a "max-elements" constraint, or a "must" expression associated with it. For performance reasons, ConfD does not check these constraints. It is assumed that the application code that generates operational data enforces the constraints.

Normally, operational data is strictly read-only. If the operational state of the device needs to be modified, it is typically done through special operations (rpc or actions in NETCONF, or special commands in the CLI). But this imposes a problem with protocols like SNMP, that do not have a mechanism to invoke arbitrary operations. In SNMP, this is solved by writing values to special objects, called *writable operational* objects. These objects are implemented in the same way as writable configuration data, described in Section 7.8, “Writable operational data”, and the section called “Writable MIB objects”.

6.2. Reading Statistics Data

A very common situation is that we wish to expose statistics data from the device. Consider for example the output of the **netstat -i** command.

```
#root netstat -i
Iface  MTU   Met   RX-OK RX-ERR RX-DRP TX-OK TX-ERR TX-DRP Flg
eth0   1500   0  212684      0      0  142470     0      0 BMRU
lo     16436  0    2077      0      0    2077     0      0 LRU
```

This is useful information to expose to the Web UI, the CLI or a management application running NETCONF.

To address this we must do two things; the statistics information must be modeled in a YANG module:

Example 6.1. netstat.yang

```
container ifaces {
  config false;
  list iface {
    key name;
    max-elements 1024;
    leaf name {
```

```
      type string;
    }
    leaf mtu {
      type uint32;
    }
    leaf metric {
      type uint64;
    }
    leaf rx_ok {
      type uint64;
    }
    leaf rx_err {
      type uint64;
    }
    leaf rx_drp {
      type uint64;
    }
    leaf tx_ok {
      type uint64;
    }
    leaf tx_err {
      type uint64;
    }
    leaf tx_drop {
      type uint64;
    }
    leaf flag {
      type string;
    }
  }
}
```

The above simple one-to-one mapping of the **netstat -i** output and a YANG data model might suffice for our needs. It can be refined later.

The second thing that must be done is to write C code that parses the **netstat -i** output. Finally we must connect that C code to ConfD. That procedure will be fully described in this chapter.

Thus we need to:

- Write a YANG module describing our operational data (see Chapter 3, *The YANG Data Modeling Language*).
- Write a mapping between the data model and the operational data as represented on the target device. The mapping is specified inside the data model itself, using callbacks to C.

6.3. Callpoints and Callbacks

The data model indicates where to invoke callbacks by annotation with `callpoints`. A callpoint has a name which later can be used by an external program to connect to that named point.

Tip

We can always define callpoints in a separate YANG module by using the `tailf:annotate` extension as described in the tailf_yang_extensions(5) manual page. This way we can keep the data model free from implementation specific details.

Assume that we wish to model the ARP table of the host:

Example 6.2. ARP table YANG module

```
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;
      }
    }
  }
}
```

The `arpe` callpoint will invoke callbacks in external programs that has registered itself with the name "arpe". The programs use the API in the `libconfd.so` library to register themselves under different callpoints.

The `config false;` statement instructs ConfD that the entire `arpentries` container is non-configuration data. Data below that point is not part of the configuration; rather it should be viewed as ephemeral read-only data.

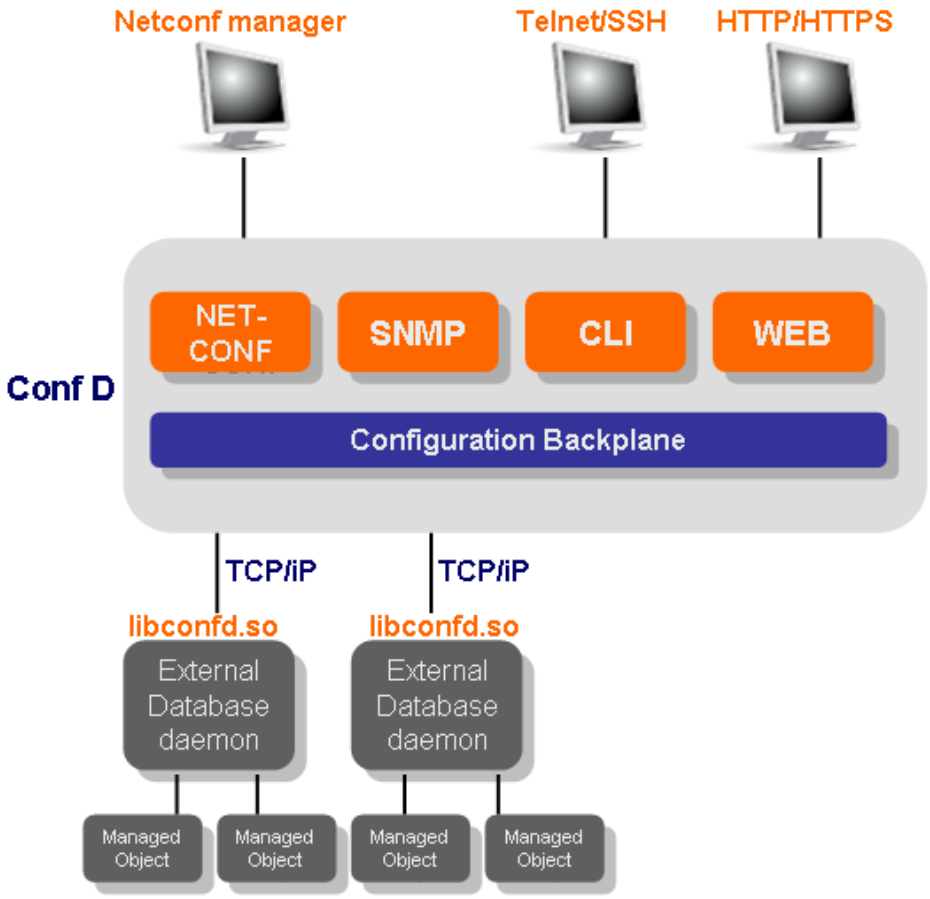
Assume we have the above YANG module loaded in ConfD. Furthermore that ConfD receives a NETCONF "get" request like:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <get/>
</rpc>
```

ConfD is configured to accept a number of `arpe` list entries contained inside an `arpentries` container. It does not know which `arpe` entries reside on the device though. With the above NETCONF request, the task for ConfD is to produce an XML structure containing all the `arpe` entries on the device.

This is solved by letting the application register itself with a set of callback C functions under the callpoint. The callback C functions do things like `get_next()`, `get_elem()` and so forth.

There can be several different C programs on the same device which register themselves under different callpoints. These C programs that register with ConfD are referred to as `daemons`.



Daemons using libconfd.so to connect to ConfD.

In the above picture we show how two separate C programs (daemons) connect to ConfD using the `libconfd.so` shared library.

6.4. Data Callbacks

Each callpoint in a YANG module must have an associated set of callback functions. The following data callback functions are required for operational data:

`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. For example, assume that the ARP table on the device looks as:

Example 6.3. Populated ARP table

```
<arpe:arpentries xmlns:arpe="http://tail-f.com/ns/example/arpe/1.0">
  <arpe:arpe>
```

```
    <arpe:ip>192.168.1.1</arpe:ip>
    <arpe:ifname>eth0</arpe:ifname>
    <arpe:hwaddr>00:30:48:88:1F:E2</arpe:hwaddr>
    <arpe:permanent>false</arpe:permanent>
    <arpe:published>false</arpe:published>
  </arpe:arpe>
  <arpe:arpe>
    <arpe:ip>192.168.1.42</arpe:ip>
    <arpe:ifname>eth0</arpe:ifname>
    <arpe:hwaddr>00:30:48:88:1F:C5</arpe:hwaddr>
    <arpe:permanent>false</arpe:permanent>
    <arpe:published>false</arpe:published>
  </arpe:arpe>
</arpe:arpentries>
```

The job of the `get_next()` callback would be to return the first key on the first invocation, namely the pair "192.168.1.1", "eth0" and then subsequently the remaining keys until there are no more keys. (The data model says that we have two keys, `ip` and `ifname`.)

`get_elem()`
: This callback is invoked by ConfD when ConfD needs to read the actual value of a leaf element. We must also implement the `get_elem()` callback for the keys. ConfD invokes `get_elem()` on a key as an existence test.

`exists_optional()`
: This callback is called for all typeless and optional elements, i.e. `presence` containers and leafs of type `empty`. For example the YANG module fragment:

```
container bs {
  presence "bs";
  config false;
  tailf:callpoint bcp;
  leaf foo {
    type string;
  }
}
```

If we do not have any typeless optional elements in our data model we need not implement this callback and can set it to NULL. A detailed description of this callback can be found in the confd_lib_dp(3) manual page.

We also have a number of additional optional callbacks that may be implemented for efficiency reasons. The precise usage of these optional callbacks is described in the man page confd_lib_dp(3).

`get_object()`
: If this optional callback is implemented, the work of the callback is to return an entire `object`, i.e. a list entry. In this case all the five elements contained in an `arpe` entry - namely the `ip`, `ifname`, `hwaddr`, `permanent` and finally `published` leafs.

`num_instances()`
: When ConfD needs to figure out how many entries we have for a list, by default ConfD will repeatedly invoke the `get_next()` callback. If this callback is registered, it will be called instead.

`get_next_object()`
: This optional callback combines `get_next()` and `get_object()` into a single callback. This callback only needs to be implemented when it is very important to be able to traverse a table fast.

`find_next()`
: This callback primarily optimizes cases where ConfD wants to start a list traversal at some other point than at the first entry of the list. It is mainly useful for lists with a large number of entries. If it is not registered, ConfD will use a sequence of `get_next()` calls to find the desired list entry.

`find_next_object()`
: This callback combines `find_next()` and `get_object()` into a single callback.

6.5. User Sessions and ConfD Transactions

In this section we will describe a number of new concepts. We will define what we mean by a user session and what ConfD transactions are. This will be further explained in Chapter 7, *The external database API*.

A user session corresponds directly to an SSH/SSL session from a management station to ConfD. A user session is associated with such data as the IP address of the management station and the user name of the user who started the session, independent of northbound agent.

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

A new transaction is started whenever an agent tries to read operational data. For each transaction two user defined callbacks are potentially invoked:

`init()`
: From the daemon's point of view, this callback will be invoked when a transaction starts. However as an optimization, ConfD will delay the invocation for a given daemon until the point where some data needs to be read, i.e. just before the first `get_next()`, `get_elem()`, etc callback.

`finish()`
: This callback gets invoked at the end of the transaction, if `init()` has been invoked. This is a good place to deallocate any local resources for the transaction. This callback is optional.

The "lazy" invocation of `init()` means that for a transaction where none of the operational data provided by a given daemon is accessed, that daemon will not have any callbacks at all invoked.

6.6. C Example with Operational Data

Assume we want to provide the state of the current ARP table on the device. To do this we need to write a YANG module which models an ARP table, and then write C functions which populates the corresponding XML tree. We use the YANG module from the previous section and save it to a file `arpe.yang` and compile the module using the **confdc** compiler as:

```
# confdc -c arpe.yang
# confdc --emit-h arpe.h arpe.fxs
```

The `--emit-h` option to confdc is used to generate a header file. Thus, in our example the generated file will be called `arpe.h`. The generated header file contains a mapping from the strings found in the data model such as `ip` or `permanent` to integer values.

Finally we must instruct ConfD where to find the newly generated schema file. Using the default ConfD configuration, ConfD looks for schema (.fxs) files under `/etc/confd`:

```
# cp arpe.fxs /etc/confd
# confd
```

After loading `arpe.fxs`, ConfD runs with the newly generated data model. Next we need to write the C program which provides the ARP data by means of C callback functions.

An actual running version of this example can be found in the `intro/5-c_stats` directory in the examples in the distribution release. We will walk through this C program here.

First we need to include `confd_lib.h` and `confd_dp.h` which are part of a ConfD release, as well as the newly generated `arpe.h`. See confdc (1) for details.

```
#include <confd_lib.h>
#include <confd_dp.h>
#include "arpe.h"
```

We use a couple of global variables as well as a structure which represents an ARP entry.

```
/* Our daemon context as a global variable */
static struct confd_daemon_ctx *dctx;
static int ctlsock;
static int workersock;


struct aentry {
    struct in_addr ip4;
    char *hwaddr;
    int perm;
    int pub;
    char *iface;
    struct aentry *next;
};

struct arpdata {
    struct aentry *arp_entries;
    struct timeval lastparse;
};
```

The struct confd_daemon_ctx *dctx is a daemon context. It is a data structure which is passed to virtually all the functions.

We are ready for the `main()` function. There we will initialize the library, connect to the ConfD daemon and install a number of callback functions as pointers to C functions. Remember the architecture of this system, ConfD executes as a common daemon, and the program we are writing executes outside the address space of ConfD. Our program links with the ConfD library (`libconfd.so`) which manages the protocol between our application and ConfD.

```
int main(int argc, char *argv[])
{
    struct sockaddr_in addr;
    int debuglevel = CONFD_TRACE;
    struct confd_trans_cbs trans;
    struct confd_data_cbs data;

    memset(&trans, 0, sizeof (struct confd_trans_cbs));
    trans.init = s_init;
    trans.finish = s_finish;

    memset(&data, 0, sizeof (struct confd_data_cbs));
    data.get_elem = get_elem;
    data.get_next = get_next;
```

```
    strcpy(data.callpoint, arpe__callpointid_arpe);

    /* initialize confd library */
    confd_init("arpe_daemon", stderr, debuglevel);

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

    if (confd_load_schemas((struct sockaddr*)&addr,
                           sizeof (struct sockaddr_in)) != CONFD_OK)
        confd_fatal("Failed to load schemas from confd\n");

    if ((dctx = confd_init_daemon("arpe_daemon")) == NULL)
        confd_fatal("Failed to initialize confdlib\n");

    /* Create the first control socket, all requests to */
    /* create new transactions arrive here */

    if ((ctlsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open ctlsocket\n");
    if (confd_connect(dctx, ctlsock, CONTROL_SOCKET, (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

    /* Also establish a workersocket, this is the most simple */
    /* case where we have just one ctlsock and one workersock */

    if ((workersock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open workersocket\n");
    if (confd_connect(dctx, workersock, WORKER_SOCKET,(struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

    if (confd_register_trans_cb(dctx, &trans) == CONFD_ERR)
        confd_fatal("Failed to register trans cb \n");

    if (confd_register_data_cb(dctx, &data) == CONFD_ERR)
        confd_fatal("Failed to register data cb \n");

    if (confd_register_done(dctx) != CONFD_OK)
        confd_fatal("Failed to complete registration \n");
```

At this point we have registered our callback functions for data manipulations under the `arpe` callpoint. Whenever data needs to manipulated below that callpoint our C callback functions should be invoked. The `confd_register_done()` call tells ConfD that we are done with the callback registrations - no callbacks will be invoked before we issue this call.

The `arpe__callpointid_arpe` symbol that is used for the `callpoint` element in the data callback registration is one of the definitions in the generated `arpe.h` file. It just maps to the string `"arpe"` that we could have used instead, but by using the symbol we make sure that if the name given with the `tailf:callpoint` statement in the YANG module is changed, without a corresponding change in the C code, the problem is detected at compile time.

We have also created one control socket and one worker socket. These are sockets owned by the application and they should be added to the `poll()` or `select()` set of the application.

All new requests that arrive from ConfD arrive on the control socket. As we will see, the `init()` callback must call the API function `confd_trans_set_fd()` which will assign a worker socket to the

transaction. All further requests and replies for this transaction will be sent on the worker socket. We can have several worker sockets and they can run in different operating system threads than the thread owning the control socket.

The poll loop could look like:

```
    while(1) {
        struct pollfd set[2];
        int ret;

        set[0].fd = ctlsock;
        set[0].events = POLLIN;
        set[0].revents = 0;

        set[1].fd = workersock;
        set[1].events = POLLIN;
        set[1].revents = 0;

        if (poll(set, sizeof(set)/sizeof(*set), -1) < 0) {
            perror("Poll failed:");
            continue;
        }

        /* Check for I/O */
        if (set[0].revents & POLLIN) {
            if ((ret = confd_fd_ready(dctx, ctlsock)) == CONFD_EOF) {
                confd_fatal("Control socket closed\n");
            } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
                confd_fatal("Error on control socket request: %s (%d): %s\n",
                     confd_strerror(confd_errno), confd_errno, confd_lasterr());
            }
        }
        if (set[1].revents & POLLIN) {
            if ((ret = confd_fd_ready(dctx, workersock)) == CONFD_EOF) {
                confd_fatal("Worker socket closed\n");
            } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
                confd_fatal("Error on worker socket request: %s (%d): %s\n",
                     confd_strerror(confd_errno), confd_errno, confd_lasterr());
            }
        }
    }
```

The crucial function above is `confd_fd_ready()`. When either of the (in this case, two) sockets from the application to ConfD are ready to read, the application is responsible for invoking the `confd_fd_ready()` function. This function will read data from the socket, unmarshal that data and invoke the right callback function with the right arguments.

We have installed two transaction callbacks: `init()` and `finish()`, and also two data callbacks: `get_next()` and `get_elem()`.

The two transaction callbacks look like:

```
static int s_init(struct confd_trans_ctx *tctx)
{
    struct arpdata *dp;

    if ((dp = malloc(sizeof(struct arpdata))) == NULL)
        return CONFD_ERR;
```

```
    memset(dp, 0, sizeof(struct arpdata));
    if (run_arp(dp) == CONFD_ERR) {
        free(dp);
        return CONFD_ERR;
    }
    tctx->t_opaque = dp;
    confd_trans_set_fd(tctx, workersock);
    return CONFD_OK;
}

static int s_finish(struct confd_trans_ctx *tctx)
{
    struct arpdata *dp = tctx->t_opaque;

    if (dp != NULL) {
        free_arp(dp);
        free(dp);
    }
    return CONFD_OK;
}
```

The `init()` callback reads the ARP table calling a function `run_arp()` and stores a local copy of a parsed ARP table in the transaction context. This data structure (struct confd_trans_ctx *tctx) is allocated by the library and used throughout the entire transaction. The `t_opaque` field in the transaction context is meant to be used by the application to store transaction local data.

A naive version of `run_arp()` could call `popen(3)` on the command **arp -an** and parse the output:

```
# arp -an
? (192.168.128.33) at 00:40:63:C9:79:FC [ether] on eth1
? (217.209.73.1) at 00:02:3B:00:3B:67 [ether] on eth0
```

The parsed ARP table created by `run_arp()` is ordered by increasing key values, since ConfD expects us to return entries in that order when traversing the list.

There may be several ConfD transactions running in parallel and some transactions may have been initiated from the CLI and the current ARP data may be stale or may be nonexistent.

The `init()` callback must also indicate to the library which socket should be used for all future traffic for this transaction. In our case, we have just one option, namely the single worker socket we created. This is done through the call to `confd_trans_set_fd()`. Also, the `init()` callback was fed a transaction context parameter. This structure is allocated by the library and fed to each and every callback function executed during the life of the transaction. The structure is defined in `confd_lib.h`.

Our `finish()` function cleans up everything.

The data callbacks look like:

```
static int get_next(struct confd_trans_ctx *tctx,
                        confd_hkeypath_t *keypath,
                        long next)
{
    struct arpdata *dp = tctx->t_opaque;
    struct aentry *curr;
    confd_value_t v[2];

    if (next == -1) {  /* first call */
        if (need_arp(dp)) {
```

```
            if (run_arp(dp) == CONFD_ERR)
                return CONFD_ERR;
        }
        curr = dp->arp_entries;
    } else {
        curr = (struct aentry *)next;
    }
    if (curr == NULL) {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }

    /* 2 keys */
    CONFD_SET_IPV4(&v[0], curr->ip4);
    CONFD_SET_STR(&v[1], curr->iface);
    confd_data_reply_next_key(tctx, &v[0], 2, (long)curr->next);
    return CONFD_OK;
}


struct aentry *find_ae(confd_hkeypath_t *keypath, struct arpdata *dp)
{
    struct in_addr ip = CONFD_GET_IPV4(&keypath->v[1][0]);
    char *iface = (char*)CONFD_GET_BUFPTR(&keypath->v[1][1]);
    struct aentry *ae = dp->arp_entries;

    while (ae != NULL) {
        if (ip.s_addr == ae->ip4.s_addr &&
            (strcmp(ae->iface, iface) == 0) )
            return ae;
        ae=ae->next;
    }
    return NULL;
}

/* Keypath example */
/* /arpentries/arpe{192.168.1.1 eth0}/hwaddr */
/*    3         2         1             0    */

static int get_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)
{
    confd_value_t v;

    struct aentry *ae = find_ae(keypath, tctx->t_opaque);
    if (ae == NULL) {
        confd_data_reply_not_found(tctx);
        return CONFD_OK;
    }
    switch (CONFD_GET_XMLTAG(&(keypath->v[0][0]))) {
    case arpe_hwaddr:
        if (ae->hwaddr == NULL) {
            confd_data_reply_not_found(tctx);
            return CONFD_OK;
        }
        CONFD_SET_STR(&v, ae->hwaddr);
        break;
    case arpe_permanent:
        CONFD_SET_BOOL(&v, ae->perm);
        break;
```

```
    case arpe_published:
        CONFD_SET_BOOL(&v, ae->pub);
        break;
    case arpe_ip:
        CONFD_SET_IPV4(&v, ae->ip4);
        break;
    case arpe_ifname:
        CONFD_SET_STR(&v, ae->iface);
        break;
    default:
        return CONFD_ERR;
    }
    confd_data_reply_value(tctx, &v);
    return CONFD_OK;
}
```

The above code needs a bit of explaining. Before doing this we need to look at how the confd_hkeypath_t data type works.

All the different data manipulation callbacks get a hashed keypath as a parameter. For example when a daemon gets invoked in `get_elem()` and ConfD wants to read the published element for a specific arp entry, the textual representation of the hkeypath is `/arpentries/arpe{1.2.3.4 eth0}/published`.

The C representation of a hashed keypath is a fixed size array of values, as in:

```
typedef struct confd_hkeypath {
    confd_value_t v[MAXDEPTH][MAXKEYLEN];
    int len;
} confd_hkeypath_t;
```

The keypath is fed in the reverse order to the application, thus - when ConfD wants to read `/arpentries/arpe{1.2.3.4 eth0}/published`, the following holds for the keypath:

- `keypath->v[0][0]` is the XML element, namely `published`.
- `keypath->v[1][0]` is the first key one step up, in our case the IP address `1.2.3.4`.
- `keypath->v[1][1]` is the second key one step up, in our case the interface name `eth0`.
- `keypath->v[2][0]` is the XML element two steps up, namely `arpe`.
- `keypath->v[3][0]` is the XML element three steps up, namely `arpentries`. The top level element. This item could also have been obtained through the expression `keypath->v[keypath->len - 1][0]`.

The actual values are represented as a union struct defined in `confd_lib.h`. The confd_value_t data type can represent all ground data types such as strings, integers, but also slightly more complex data types such as IP addresses and the various date and time data types found in XML schema.

`confd_lib.h` defines a set of macros to set and get the actual values from confd_value_t variables. For example this code sets and gets an individual value:

```
confd_value_t myval;
int i = 99;

CONFD_SET_INT32(&myval, i);
```

```
assert(99 == CONFD_GET_INT32(&myval));
```

One important variant of confd_value_t is string. All data values which are of type `string`, or of a type derived from `string`, are passed from ConfD to the application as NUL terminated strings. Thus confd_value_t contains a length indicator *and* is NUL terminated.

All strings consist of an unsigned char* pointer and a length indicator. To copy such a string into a local buffer we need to write code like:

```
char *mybuf = malloc(CONFD_GET_BUFSIZE(someval)+1);
strcpy(mybuf, (char*)CONFD_GET_BUFPTR(someval));
```

On the other hand, when the application needs to reply with a string value to ConfD, the application can choose to use either a NUL terminated string or a buffer with a length indicator using the following macros:

```
confd_value_t myval;
CONFD_SET_STR(&myval, "Frank Zappa");
```

or

```
confd_value_t myval;
CONFD_SET_BUF(&myval, buf, buflen);
```

XML tags are also represented as confd_value_t. Remember that we said that `keypath->v[0][0]` was the actual XML element. Also remember that **confdc** generated a .h file. The `arpe.h` file, containing all the XML elements from `arpe.yang` as integers. The following code uses that to switch on the XML tag:

```
    switch (CONFD_GET_XMLTAG(&(keypath->v[0][0]))) {
    case arpe_hwaddr:
        if (ae->hwaddr == NULL) {
            confd_data_reply_not_found(tctx);
            return CONFD_OK;
        }
        CONFD_SET_STR(&v, ae->hwaddr);
        break;
    case arpe_permanent:
        CONFD_SET_BOOL(&v, ae->perm);
        break;
```

Each keypath has a textual representation, so we can format a keypath by means of the API call `confd_pp_kpath()`. A hashed keypath, a confd_hkeypath_t, represents a unique path down through the XML tree and it is easy and efficient to walk the path through `switch` statements since the individual XML elements in the path are integers.

The purpose of both functions, (`get_next()` and `get_elem()`), is to return data back to ConfD. Data is not returned explicitly through return values from the callback functions, but rather through explicit API calls.

So when the application gets invoked in `get_elem()` via a call to `confd_fd_ready()`, we need to return a single value to ConfD. We do this through the call to `confd_data_reply_value()`. Thus the following code snippet returns an integer value to ConfD.

```
confd_value_t myval;
CONFD_SET_INT32(&myval, 7777);
confd_data_reply_value(tctx, &myval);
```

The `get_elem()` callback is also used as an existence test by ConfD. It may seem redundant to implement the `get_elem()` callback for a keypath such as: `"/arpentries/arpe{1.2.3.4 eth0}/ip"` since the only possible reply can be the IP address "1.2.3.4" which is already part of the keypath. However, the user can enter any random path in the CLI and ConfD uses the `get_elem()` callback to check whether an entry exists or not.

If the entry does not exist, the callback should call `confd_data_reply_not_found()` and then return CONFD_OK. This is not an error.

The API is fully documented in the confd_lib_dp(3) manual page.

The `get_next()` callback gets invoked when ConfD needs to read all the keys of a certain list such as our ARP entries. The *`next`* parameter will have the value -1 on the first invocation to `get_next()`. This invocation needs to return the first key in the ordered list created by `run_arp()`. In our case, with our ARP entries, we have multiple keys. According to the data model the pair of the interface name and the IP address makes up the key. Thus we need to return two values:

```
    CONFD_SET_IPV4(&v[0], curr->ip4);
    CONFD_SET_STR(&v[1], curr->iface);
    confd_data_reply_next_key(tctx, &v[0], 2, (long)curr->next);
```

The last parameter to `confd_data_reply_next_key()` is a long integer which will be fed to us as the *`next`* parameter on the subsequent call. We cast the pointer to the next struct aentry* as a long.

In the above code, we registered a single set of callback C functions on the callpoint. Sometimes we may have different daemons that handle different kinds of data, but under the same callpoint. Say for example that we have a list of interfaces, VLAN interfaces and regular interfaces. We have different software modules which handle the VLAN interfaces and the regular interfaces. In this case, we may use `confd_register_range_data_cb()` (See confd_lib_dp(3)) which makes it possible to install a set of callbacks on a range of keys, for example one set of callbacks for `eth0` to `ethX` and another set of callbacks in the range from `vlan0` to `vlanX`.

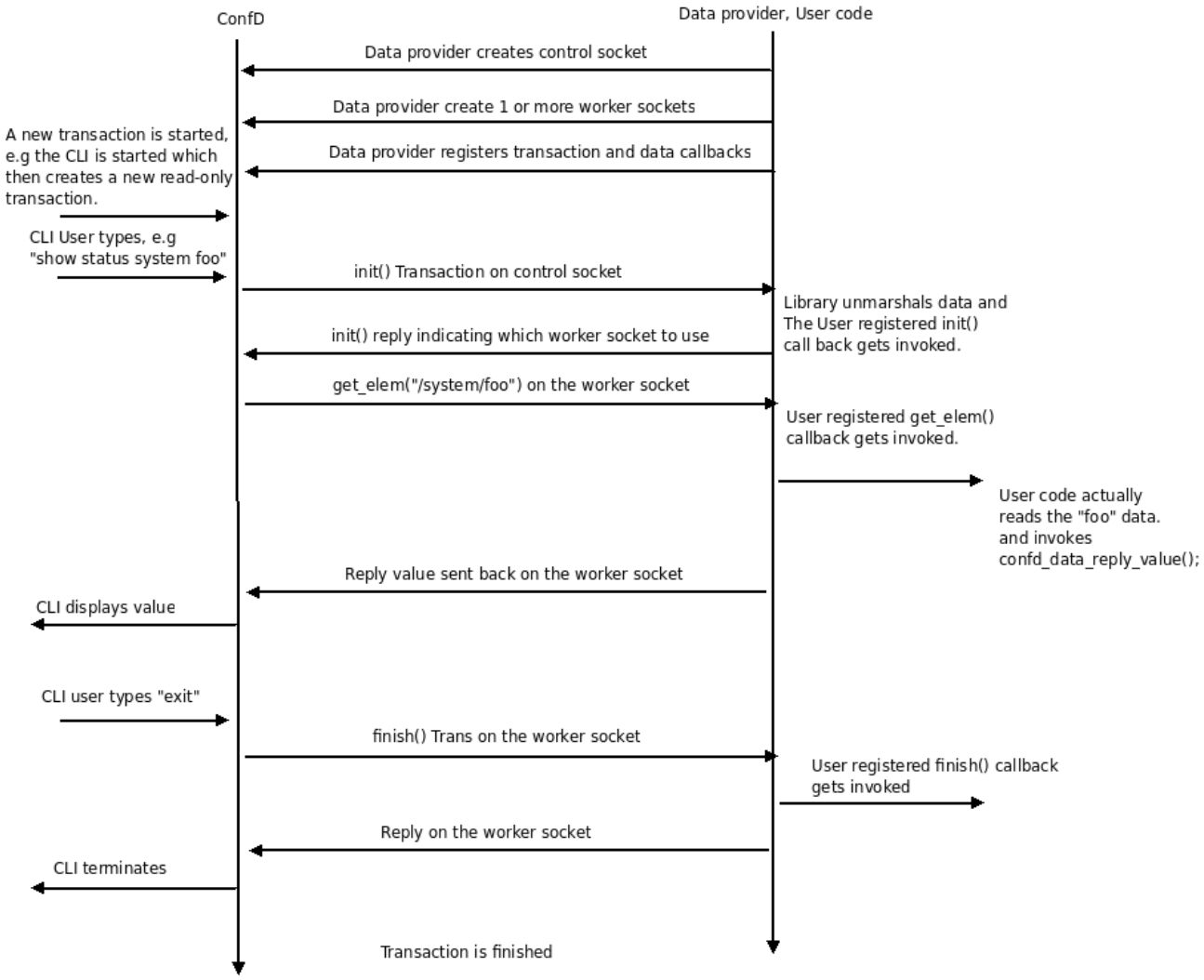
Also notable is the consistency of the data. If we use CDB to store our configuration data and we use this external data API to deliver statistics data which is volatile we must choose whether we want to deliver an exact snapshot of the statistics data or not. ConfD will consecutively call `get_next()` to gather all the keys for a set of dynamic elements. A moment later ConfD will invoke `get_elem()` or `get_object()` to gather the actual data. If this data no longer exists, the application can invoke `confd_data_reply_not_found()` and all is fine.

An alternative for the application if we must always return consistent snapshots, is to gather and buffer all the data in the `init()` callback and then return both `get_next()` data as well as `get_elem()` data from those internal data structures. This data can be stored in the `t_opaque` field in the transaction context and be released in the `finish()` callback.

In our example code above we have chosen the latter approach. Also notable is the check for age of data at the beginning of `get_next()`. A NETCONF transaction is typically short lived, whereas a CLI transaction remains live for as long as the user is logged in. Thus we may have to refresh the locally stored ARP table if it is deemed to be too old.

6.7. The Protocol and a Library Threads Discussion

We start this section with a picture showing the sequence of events that occur when the user defined callback functions get invoked by the library.



Event chain that triggers user callbacks

On the library side we have one control socket and one or more worker sockets. The idea behind this architecture is that it shall be possible to have a software architecture, whereby a main thread owns the control socket, and we also have a set of worker threads, each owning one or more worker sockets. A request to execute something arrives from ConfD on the control socket, and the thread owning the control socket, the main thread, can then decide to assign a worker thread for that particular activity, be it a validation, a new transaction or the invocation of an action. The owner of the control socket must thus have a mapping between worker sockets and thread workers. This is up to the application to decide.

The downside of the architecture proposed above is complexity, whereas the upside is that regardless of how long time it takes to execute an individual request from ConfD, the data provider is always ready to accept and serve new callback requests from ConfD.

The case for a multi threaded dataprovider maybe isn't as strong as one could think. Say that we have a statistics data provider which lists a very long list of statistics items, e.g. a huge routing table. If a CLI user invokes the command to show all routing table entries, there will be a long series of `get_next()` and `get_elem()` callback invocations. As long as the application is still polling the control socket, other northbound agents can very well sneak in and execute their operations *while* the routing table is being displayed. For example another CLI user issuing a request to reboot the host, will get his reboot request served at the same time as the first CLI user is displaying the large routing table.

A data provider with just one thread, one control socket and one worker socket will never hang longer than it takes to execute a single callback invocation, e.g. a single invocation of `get_elem()`, `validate()` or `action()`. In many cases it will still be a good design to use at least one thread for the control socket and one for the worker socket - this will allow for control socket requests to be handled quickly even if the data callbacks require more processing time. If we have long-running action callbacks (e.g. file download), multi-threading may be essential, see Section 11.2.2, “Using Threads”.

The `intro/9-c_threads` example in the ConfD examples collection shows one way to use multi-threading in a daemon that implements both operational data callbacks and action callbacks. It has one thread for the control socket and only a single worker socket/thread for the data callbacks, while multiple worker sockets/threads are used to handle the action callbacks.

When we use multiple threads, it is important to remember that threads can not "share" socket connections to ConfD. For the data provider API, this is basically fulfilled automatically, as we will not have multiple threads polling the same socket. But when we use e.g. the CDB or MAAPI APIs, the application must make sure that each thread has its own sockets. I.e. the ConfD API functions are thread-safe as such, but multiple threads using them with the same socket will have unpredictable results, just as multiple threads using the read() and write() system calls on the same file descriptor in general will. In the ConfD case, one thread may end up getting the response to a request from another, or even a part of that response, which will result in errors that can be very difficult to debug.

6.8. Operational data in CDB

It is possible to use CDB to store not only the configuration data but also operational data. Depending on the application and the underlying architecture it may be easier for some of the managed objects to write their operational data into CDB. Depending on the type of data, this would typically be done either at regular intervals or whenever there is a change in the data. If this is done, no instrumentation functions need to be written. The operational data then resides in CDB and all the northbound agents can read the operational data automatically from CDB.

Similar to the CDB read interface, we need to create a CDB socket and also start a CDB session on the socket before we can write data

The necessary steps are:

1. `cdb_connect()`
2. `cdb_start_session()` followed by `cdb_set_namespace()`
3. A series of calls to one or several of the CDB set functions, `cdb_set_elem()`, `cdb_create()`, `cdb_delete()` `cdb_set_object()` or `cdb_set_values()`

   These functions are described in detail in the confd_lib_cdb(3) manual page.
4. 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()`, see the confd_lib_cdb(3) manual page. A command line utility called **confd_load** can also be used, see confd_load(1).

We use the `tailf:cdb-oper` statement to indicate that operational data should be stored in CDB, see the tailf_yang_extensions(5) manual page. The data can be either persistent, i.e. stored on disc, or volatile, i.e. stored in RAM only - this is controlled by the `tailf:persistent` substatement to `tailf:cdb-oper`.

As a realistic example we model IP traffic statistics in a Linux environment. We have a list of interfaces, stored in CDB and then for each interface we have a statistics part. This example can be found in `cdb_oper/ifstatus` in the examples collection. This is what our data model looks like:

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

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


  container interfaces {
    list interface {
      key name;
      max-elements 1024;
      leaf name {
        type string;
      }
      list address {
        key name;
        max-elements 64;
        leaf name {
          type inet:ipv4-address;
        }
        leaf prefix-length {
          type int32;
          mandatory true;
        }
      }
      container status {
        config false;
        tailf:cdb-oper;

        container receive {
          leaf bytes {
            type uint64;
            mandatory true;
          }
          leaf packets {
            type uint64;
            mandatory true;
          }
          leaf errors {
            type uint32;
            mandatory true;
          }
          leaf dropped {
            type uint32;
            mandatory true;
          }
        }
        container transmit {
          leaf bytes {
            type uint64;
            mandatory true;
```

```
          }
          leaf packets {
            type uint64;
            mandatory true;
          }
          leaf errors {
            type uint32;
            mandatory true;
          }
          leaf dropped {
            type uint32;
            mandatory true;
          }
          leaf collisions {
            type uint32;
            mandatory true;
          }
        }
      }
    }
  }
}
```

Note the element `/interfaces/interface/status`, it has the substatement `config false;` and below it we find a `tailf:cdb-oper;` statement. If we had implemented this operational data using the techniques from the previous sections in this chapter, we would have had to write instrumentation callback functions for the above operational data. For example `get_elem()` which would then be given a path, e.g. `/interfaces/interface{eth0}/status/receive/bytes` When we use the `tailf:cdb-oper;` statement these instrumentation callbacks are automatically provided internally by ConfD. The downside is that we must populate the CDB data from the outside.

A function which reads network traffic statistics data and updates CDB according to the above data model is:

```
#define GET_COUNTER() {                          \
        if ((p = strtok(NULL, " \t")) == NULL)  \
            continue;                            \
        counter = atoll(p);                      \
    }

static int update_status(int sock)
{
    FILE *proc;
    int ret;
    char buf[BUFSIZ];
    char *ifname, *p;
    long long counter;
    confd_value_t val[1 + 4 + 1 + 5];
    int i;

    if ((ret = cdb_start_session(sock, CDB_OPERATIONAL)) != CONFD_OK)
        return ret;
    if ((ret = cdb_set_namespace(sock, if__ns)) != CONFD_OK)
        return ret;

    if ((proc = fopen("/proc/net/dev", "r")) == NULL)
        return CONFD_ERR;
    while (ret == CONFD_OK && fgets(buf, sizeof(buf), proc) != NULL) {
        if ((p = strchr(buf, ':')) == NULL)
            continue;
```

```
        *p = ' ';
        if ((ifname = strtok(buf, " \t")) == NULL)
            continue;

        i = 0;

        CONFD_SET_XMLTAG(&val[i], if_receive, if__ns); i++;
        GET_COUNTER();          /* rx bytes */
        CONFD_SET_UINT64(&val[i], counter); i++;
        GET_COUNTER();          /* rx packets */
        CONFD_SET_UINT64(&val[i], counter); i++;
        GET_COUNTER();          /* rx errs */
        CONFD_SET_UINT32(&val[i], counter); i++;
        GET_COUNTER();          /* rx drop  */
        CONFD_SET_UINT32(&val[i], counter); i++;
        /* skip remaining rx counters */
        GET_COUNTER(); GET_COUNTER(); GET_COUNTER(); GET_COUNTER();

        CONFD_SET_XMLTAG(&val[i], if_transmit, if__ns); i++;
        GET_COUNTER();          /* tx bytes */
        CONFD_SET_UINT64(&val[i], counter); i++;
        GET_COUNTER();          /* tx packets */
        CONFD_SET_UINT64(&val[i], counter); i++;
        GET_COUNTER();          /* tx errs */
        CONFD_SET_UINT32(&val[i], counter); i++;
        GET_COUNTER();          /* tx drop  */
        CONFD_SET_UINT32(&val[i], counter); i++;
        GET_COUNTER();          /* skip */
        GET_COUNTER();          /* tx colls */
        CONFD_SET_UINT32(&val[i], counter); i++;

        ret = cdb_set_object(sock, val, i,
                             "/interfaces/interface{%s}/status", ifname);
        if (ret == CONFD_ERR && confd_errno == CONFD_ERR_BADPATH)
            /* assume interface doesn't exist in config */
            ret = CONFD_OK;
    }
    fclose(proc);

    cdb_end_session(sock);

    return ret;
}
```

We typically call this function at regular intervals.

6.9. Delayed Replies

If the data source is communicated with through some means of IPC it may be inconvenient to hang in the callback functions and wait for the reply from the data source. The solution to this problem is to return a special return value from the callback and then later explicitly send the response once it is available.

All the transaction callbacks as well as all the data callbacks can optionally return the value `CONFD_DELAYED_RESPONSE`. This means that the callback returns, and we typically end up in our main poll loop again. Once the reply returns it is then up to the application to send the reply back to ConfD.

The `libconfd` library contains a number of routines that can be invoked to convey a delayed response. The callbacks are divided in two groups. The first group is the one where the actual return value from the

callback is the value that is sent to ConfD as a response. A good example is the the transaction `init()` callback or the the data callback `set_elem()`. In both these case if the callback returns `CONFD_OK` a positive ack is sent back to ConfD by the library. If we instead return `CONFD_DELAYED_RESPONSE` the application must - once the reply is available - use either of the functions `confd_delayed_reply_ok()` or `confd_delayed_reply_error()` to explicitly send the reply. If no reply is sent within 120 seconds (configurable through `confd.conf`) the data provider is considered dead by ConfD and ConfD will close all sockets to the data provider.

Another group of callbacks are the callbacks that require the application to explicitly send a reply back to ConfD before returning. A good example is the data callback `get_elem()`. The application must explicitly call `confd_data_reply_value()` before returning - unless the `CONFD_DELAYED_RESPONSE` value is returned. If so, it is up to the application to later, when the response value is available, explicitly call the `confd_data_reply_value()` function to send back the return value.

6.10. Caching Operational Data

For operational data handled by an external data provider (i.e., using `tailf:callpoint`), the values of elements may be kept for a certain time in a cache in ConfD. If such an element is accessed, its value will be taken from the cache, and the data provider not called.

The cache is enabled, and the default time to keep values in the cache configured, with the element `/confdConfig/opcache` in the `confd.conf` file, for example:

```
  <opcache>
    <enabled>true</enabled>
    <timeout>5</timeout>
  </opcache>
```

By default, the cache is disabled. The timeout value is given in seconds, it does not have a default. If **confd --reload** is done, the cache will use the new timeout value. If the cache is disabled, the stored values are cleared.

To indicate that the elements handled by a callpoint are to be saved in the cache, use the `tailf:cache` statement:

```
leaf packetCounter {
  type uint64;
  config false;
  tailf:callpoint a1 {
    tailf:cache true;
  }
}
```

It is also possible to override the cache timeout specified in `confd.conf` by using the `tailf:timeout` substatement with `tailf:cache` in the data model.

```
leaf packetCounter {
  type uint64;
  config false;
  tailf:callpoint a1 {
    tailf:cache true {
      tailf:timeout 7;
    }
  }
}
```

The timeout specified this way will be used for the node with the `tailf:timeout` statement and any descendants of that node, unless another `tailf:cache` statement is used on a descendant node. Using

`tailf:cache` without a `tailf:timeout` substatement will cause the timeout to revert to the one specified in `confd.conf`.

The results of `get_next()` and `find_next()` operations can not be cached in general, since the *`next`* value returned by the data provider does not necessarily identify a specific list entry (e.g. it could be a fixed pointer to a data structure holding the "next entry" information). However in the special case that the data provider returns `-1` for *`next`* the result can be cached, since retrieval of the next entry will then use a `find_next` operation with the complete set of keys from the previous entry. See the confd_lib_dp(3) manual page for further details.

The cache can be cleared, partially or completely, by means of the `maapi_clear_opcache()` function - see the confd_lib_maapi(3) manual page.

6.11. Operational data lists without keys

It is possible to define lists for operational data without any keys in the YANG data model, e.g.:

```
list memory-pool {
  config false;
  tailf:callpoint memstats;
  leaf buffer-size {
    type uint32;
  }
  leaf number-of-buffers {
    type uint32;
  }
}
```

To support this without having completely separate APIs, we use a "pseudo" key in the ConfD APIs for this type of list. This key is not part of the data model, and completely hidden in the northbound agent interfaces, but is used with e.g. the `get_next()` and `get_elem()` callbacks as if it were a normal key.

This "pseudo" key is always a single signed 64-bit integer, i.e. the confd_value_t type is `C_INT64`. The values can be chosen arbitrarily by the application, as long as a key value returned by `get_next()` can be used to get the data for the corresponding list entry with `get_elem()` or `get_object()` as usual. It could e.g. be an index into an array that holds the data, or even a memory address in integer form.

There are some issues that need to be considered though:

- In some cases ConfD will do an "existence test" for a list entry. For "normal" lists, this is done by requesting the first key leaf via `get_elem()`, but since there are no key leafs, this can not be done. Instead ConfD will use the `exists_optional()` callback for this test. I.e. a data provider that has this type of list must implement this callback, and handle a request where the keypath identifies a list entry.

- In the response to the `get_next_object()` callback, the data provider is expected to provide the key values along with the other leafs in an array that is populated according to the data model. This must be done also for this type of list, even though the key isn't actually in the data model. The "pseudo" key must always be the first element in the array, and for the `confd_data_reply_next_object_tag_value_array()` reply function, the tag value 0 should be used. Note that the key should *not* be included in the response to the `get_object()` callback.

- The same approach is used when we store operational data in CDB - the path used in the write (and read) functions in the CDB API must include the "pseudo" integer key. If multiple list entries are to be written with a single call to `cdb_set_values()`, which takes a tagged value array, the key for

each entry must be included in the array with a tag value of 0, in the same way as described above. This applies also to reading multiple entries with a single call to `cdb_get_values()`.

Chapter 7. The external database API

7.1. Introduction to external data

In the previous chapter we showed how to use ConfD with read-only operational data. In this chapter we will use the same APIs from `libconfd.so` to implement externally stored configuration data. This is the opposite of CDB, if CDB is used to store the configuration data, this section can be skipped.

We show how ConfD can use an external database as data source. The external data base can either be a full-fledged real data base or something as simple as a text file.

The configuration of the network device is modeled by a YANG module. It describes the data model of the device and ConfD needs to populate the XML data tree with actual data.

If the ConfD built-in XML database (CDB) is used to hold all configuration data, ConfD will automatically read and write into that database. If, on the other hand, the actual configuration data is kept outside of ConfD we need user supplied code to provide ConfD with the actual data of the configuration.

7.2. Scenario - The database is a file

Many standard UNIX applications read their configuration from a static file. If we want to integrate such an application into our network device, it may not be feasible to rewrite the application so that it reads its configuration from the device configuration database. In general we want to change the code of the application as little as possible.

Examples of such applications are abundant. In general this applies to all open source applications generally found on UNIX machines.

In order to integrate such an application into ConfD we must first write a YANG module which models the part of the application (the part of the application's configuration file) which we wish to be able to configure. Following that we must write C code which can read, parse, manipulate and write the configuration file in question and finally we must connect that C code to ConfD.

We did precisely this exercise in Chapter 5, *CDB - The ConfD XML Database*, however the solution from that chapter had the actual configuration data in CDB, and the configuration file was generated. Thus, if the file was edited or otherwise changed externally, those changes would be overwritten the next time we regenerated the file. In this chapter we will show how to use the actual file as a database. I.e. no configuration data is ever kept inside ConfD, the data resides outside ConfD.

7.3. Callpoints and callbacks

Similar to how we managed operational data, we need to define a data model and annotate the model with a callpoint.

Assume that we wish to model a set of 'server' structures as in the following YANG module:

Example 7.1. A list of server structures

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

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

  /* A set of server structures */
  container servers {
    tailf:callpoint simplecp;
    list server {
      key name;
      max-elements 64;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ipv4-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
    }
  }
}
```

The callpoint called `simplecp` instructs ConfD that whenever it needs to populate the XML tree below `simplecp`, it must invoke callbacks in an external program which has registered itself with the name `simplecp`. The external programs use the API in `libconfd.so` to register themselves under different callpoints.

7.4. Data Callbacks

When we implemented the operational data callbacks we had to implement a set of callbacks for each callpoint. With external data we must do the same, but some additional callbacks must also be implemented. The data callbacks `get_next()`, `get_elem()`, `get_object()`, `get_next_object()`, `find_next()`, `find_next_object()`, `num_instances()`, and finally `exists_optional()` work precisely the same for external data as they do for operational data. Those callbacks are thus described in the previous chapter.

Additionally the following data callback functions are required for external data:

- `create()` - This callback creates a new list entry. In the case of the `smp.yang` module above, this function needs to create a new empty "server" entry. Once the entry is created, it will be populated with values through a series of calls to `set_elem()`.
- `remove()` - This callback needs to remove an entire list entry and all its subelements.
- `set_elem()` - This callback sets the value of a leaf.

7.5. User sessions and ConfD Transactions

Again, similar to the chapter on operational data user sessions are created when a user logs in, and new transactions are created when an agent initiates an activity.

If we deal with operational data, the different phases are not interesting, thus then we only had to implement the `init()` and a `finish()` callback. This section describes the states of a ConfD transaction and also which user callbacks that need to be implemented in order to participate in the transaction.

In a device where ConfD is used to manage the configuration data there can be multiple sources of data. To use ConfD terminology: there can be several different daemons that connect to ConfD under different callpoints. Some callpoints may also be served by CDB.

Furthermore, a set of write operations may involve several of these daemons as well as CDB. In order to ensure that all participants perform the operations, ConfD orchestrates a two-phase commit protocol towards the different participants. Each NETCONF operation, such as `edit-config` or each call to **commit** in the CLI will be clumped into a ConfD transaction. If we store our data outside of ConfD - as will be described in this chapter - we must implement a number of callback functions in order to participate in the various states of the transaction.

An individual daemon may (or may not) implement the callbacks for the two-phase commit protocol. If there is only one daemon and CDB is not used at all, the two-phase commit protocol may be skipped. The reason for this is that when there is only one participant, the two-phase commit protocol is irrelevant.

Each NETCONF operation, i.e. each `edit-config` and so forth, will execute as one transaction. Thus transactions originating from NETCONF will be fairly short-lived entities whereas transactions originating from the CLI or the Web UI will be longer.

A daemon that wishes to participate in the two-phase commit transaction must implement a number of callback functions.

- `init()` - As for operational data, from the daemon's point of view the `init()` callback is invoked when a transaction starts, but ConfD delays the actual invocation as an optimization. For a daemon providing configuration data, `init()` is invoked just before the first data-reading callback, or just before the `trans_lock()` callback (see below), whichever comes first. When a transaction has started, it is in a state we refer to as `READ`. ConfD will, while the transaction is in the `READ` state, execute a series of read operations towards (possibly) different callpoints in the daemon.

  Any write operations performed by the management station are accumulated by ConfD and the daemon doesn't see them while in the `READ` state.

- `trans_lock()` - This callback gets invoked by ConfD at the end of the transaction. ConfD has accumulated a number of write operations and will now initiate the final write phases. Once the `trans_lock()` callback has returned, the transaction is in the `VALIDATE` state. In the `VALIDATE` state, ConfD will (possibly) execute a number of read operations in order to validate the new configuration. Following the read operations for validations comes the invocation of one of the `write_start()` or `trans_unlock()` callbacks.

- `trans_unlock()` - This callback gets invoked by ConfD if the validation failed or if the validation was done separate from the commit (e.g. by giving a **validate** command in the CLI). Depending on where the transaction originated, the behavior after a call to `trans_unlock()` differs. If the transaction originated from the CLI, the CLI reports to the user that the configuration is invalid and the transaction remains in the `READ` state whereas if the transaction originated from a NETCONF client, the NETCONF operation fails and a NETCONF `rpc` error is reported to the NETCONF client/manager.

- `write_start()` - If the validation succeeded, the `write_start()` callback will be called and the transaction enters the `WRITE` state. While in `WRITE` state, a number of calls to the write callbacks `set_elem()`, `create()` and `remove()` will be performed.

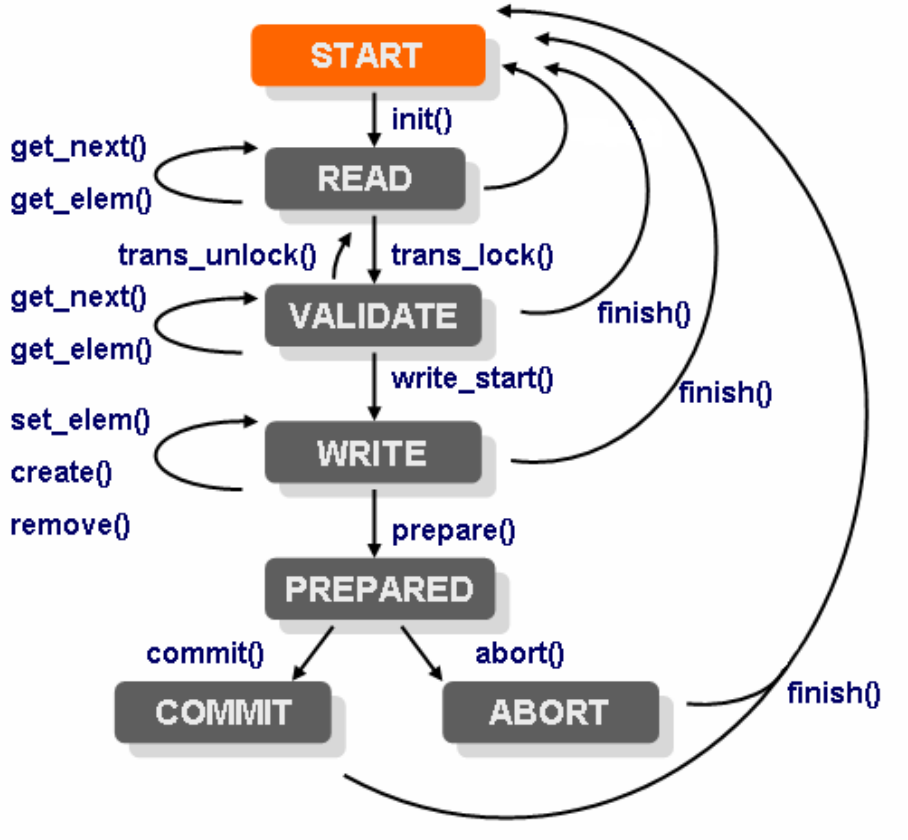
  If the underlying database supports real atomic transactions, this is a good place to start such a transaction.

The application should not modify the real running data here. If, later, the `abort()` callback is called, all write operations performed in this state must be undone.

- `prepare()` - Once all write operations are executed, the `prepare()` callback is executed. This callback ensures that all participants have succeeded in writing all elements. The purpose of the callback is merely to indicate to ConfD that the daemon is ok, and has not yet encountered any errors.
- `abort()` - If any of the participants return an error or fail to reply in the `prepare()` callback, the remaining participants all get invoked in the `abort()` callback. All data written so far in this transaction should be disposed of.
- `commit()` - If all participants successfully replied in their respective `prepare()` callbacks, all participants get invoked in their respective `commit()` callbacks. This is the place to make all data written by the write callbacks in `WRITE` state permanent.
- `finish()` - And finally, the `finish()` callback gets invoked at the end. This is a good place to deallocate any local resources for the transaction.

  The `finish()` callback can be called from several different states.

The following picture illustrates the conceptual state machine a ConfD transaction goes through.



ConfD transaction state machine

All callbacks except the `init()` callback are optional. If a callback is not implemented, it is the same as a succeeding empty implementation such as:

```
int mycallback(struct confd_trans_ctx *tctx)
{
```

```
    return CONFD_OK;
}
```

In the following examples, we will initially not use these transactions at all. We will implement the `init()` callback only and let the other transaction callbacks be `NULL`.

7.6. External configuration data

In this section we provide a commented example which manages actual configuration data. The idea is that ConfD runs the NETCONF agent and is entirely responsible for the candidate configuration and possibly runs the CLI and the Web UI. The application is responsible for maintaining and storing the configuration data.

An actual running version of this example can be found in the examples directory of a ConfD release under `user_guide_examples/simple_no_trans`.

The example system stores "servers" with name, ip, and port on a file. Our YANG module will be very simple; we have:

Example 7.2. The smp.yang module

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

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

  /* A set of server structures */
  container servers {
    tailf:callpoint simplecp;
    list server {
      key name;
      max-elements 64;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ipv4-address;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
    }
  }
}
```

To implement this we first need a small database. We choose to use a simple array of "server" structures, as in:

```
struct server {
    char name[256];
```

```
    struct in_addr ip;
    unsigned int port;
};

static struct server running_db[64];
static int num_servers = 0;
```

To create a new "server" in the database we add a new server structure to the array, as in:

```
static struct server *add_server(char *name)
{
    int i, j;

    for (i=0; i < num_servers; i++) {
        if (strcmp(running_db[i].name, name) > 0) {
            /* found the position to add at, now shuffle the */
            /* remaining elems in the array one step */
            for (j = num_servers; j > i; j--) {
                running_db[j] = running_db[j-1];
            }
            break;
        }
    }
    num_servers++;
    memset(&running_db[i], 0, sizeof(struct server));
    strcpy(running_db[i].name, name);
    return &running_db[i];
}

static struct server *new_server(char *name, char *ip, char *port)
{
    struct server *sp = add_server(name);
    sp->ip.s_addr = inet_addr(ip);
    sp->port = atoi(port);
    return sp;
}
```

We keep the array ordered according to the key (server name), since ConfD expects us to return entries in that order when traversing the list.

Note that at first glance this code looks like we may write off the end of the `running_db` array. But this is not the case, since the `server` list in the data model is defined with `max-elements 64;`. This means that ConfD will guarantee that there are never more than 64 servers.

To search the database for a specific server we have:

```
/* Find a specific server */
static struct server *find_server(confd_value_t *v)
{
    int i;
    for (i=0; i < num_servers; i++) {
        if (confd_svcmp(running_db[i].name, v) == 0)
            return &running_db[i];
    }
    return NULL;
}
```

Our `find_server()` function utilizes a `strcmp()`-like function from `libconfd.so` - the function `confd_svcmp()` compares a string char* value to a confd_value_t value. The type of the confd_value_t must obviously be either a string or a buffer.

The initialization code is very similar to the ARP example in the chapter on operational data, with the exception that we must also here register functions to write new data. We need to register callbacks to `set_elem()` which set the value of a leaf element such as `/servers/server{www}/ip`. We also need to register callback functions that can create a new "server" entry and delete old "server" entries. Thus we initialize our data callback structure struct confd_data_cbs as:

```
data.get_elem = get_elem;
data.get_next = get_next;
data.set_elem = set_elem;
data.create   = create;
data.remove   = doremove;
```

The `get_elem()` and `get_next()` callbacks can be implemented in a manner similar to how we implemented the corresponding callbacks for the ARP example. For example:

Example 7.3. get_next() callback for smp.yang

```
static int get_next(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    long next)
{
    confd_value_t v;

    if (next == -1) { /* Get first key */
        if (num_servers == 0) {  /* Db is empty */
            confd_data_reply_next_key(tctx, NULL, -1, -1);
            return CONFD_OK;
        }
        CONFD_SET_STR(&v, running_db[0].name);
        confd_data_reply_next_key(tctx, &v, 1, 1);
        return CONFD_OK;
    }
    if (next == num_servers) {  /* Last elem */
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }
    CONFD_SET_STR(&v, running_db[next].name);
    confd_data_reply_next_key(tctx, &v, 1, next+1);
    return CONFD_OK;
}
```

The create callback is easy. The keypath passed to the `create()` callback will have the new key (last in the string) as first element (in the array). Recall that the keypaths are passed in reversed order. For example when ConfD wants to create a new server entry, named to for example "smtp", the keypath will look like `/servers/server{smtp}`.

The data model can optionally specify default values. In `smp.yang` we didn't use that feature. For example the "port" leaf was specified as:

```
leaf port {
  type inet:port-number;
  mandatory true;
}
```

and not as

```
leaf port {
```

```
  type inet:port-number;
  default 0;
}
```

Our C code needs to be able to create list entries in the database without any of the actual values of the leafs given. All keys will be given but none of the actual values of the other leafs (except for the key leafs). ConfD will set all the missing values using the `set_elem()` callback. Our `create()` callback looks like:

Example 7.4. create() callback for smp.yang

```
static int create(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    confd_value_t *key = &keypath->v[0][0];
    add_server((char *)CONFD_GET_BUFPTR(key));
    return CONFD_OK;
}
```

In a similar manner, the `remove()` callback deletes a server entry.

Example 7.5. remove() callback for smp.yang

```
static int doremove(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    int i, j;
    confd_value_t *key = &keypath->v[0][0];

    for (i=0; i < num_servers; i++) {
        if (confd_svcmp(running_db[i].name, key) == 0) {
            /* found the elem to remove, now shift the */
            /* remaining elems in the array one step */
            for (j=i+1; j < num_servers; j++) {
                running_db[j-1] = running_db[j];
            }
            num_servers--;
            return CONFD_OK;
        }
    }
    return CONFD_OK;
}
```

Finally here is the `set_elem()` callback which is responsible for setting a leaf value. The code is:

Example 7.6. set_elem() callback for smp.yang

```
static int set_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    confd_value_t *newval)
{
    confd_value_t *tag = &(keypath->v[0][0]);
    struct server* s = find_server(&(keypath->v[1][0]));
    if (s == NULL) {
        confd_trans_seterr(tctx, "no such server found");
        return CONFD_ERR;
    }
```

```
    switch (CONFD_GET_XMLTAG(tag)) {
    case smp_ip:
        s->ip = CONFD_GET_IPV4(newval);
        break;
    case smp_port:
        s->port = CONFD_GET_INT32(newval);
        break;
    default:
        return CONFD_ERR;
    }
    return CONFD_OK;
}
```

Note that there is no `switch` clause for `smp_name` - ConfD will never change key values by invoking `set_elem()` for key leafs. Changing keys can only be done by a combination of `remove()` and `create()` invocations, followed by `set_elem()` invocations for the non-leaf keys in the created list entry.

7.7. External configuration data with transactions

In this section we introduce and use the transaction callbacks.

An actual running version of this example can be found in the examples directory of a ConfD release under `user_guide_examples/simple_trans`.

An application is invoked in `trans_lock()` when a transaction is committed or when a transaction is validated (e.g. by doing **validate** in the CLI), and the transaction enters the `VALIDATE` state.

When the application is invoked in the `trans_lock()` callback, the following is guaranteed.

- A sequence of callbacks will be invoked without delays. ConfD has accumulated a number of `write()` operations and will execute them in a sequence without delays.
- No callbacks to any other transactions towards the same data store will be executed between the invocation of `trans_lock()` and the invocation of `finish()` (or `trans_unlock()`). Thus all transactions towards a given data store are serialized once they reach the `VALIDATE` state.

After validation, either `trans_unlock()` or `write_start()` is invoked. `trans_unlock()` is called when the transaction is validated only, and `write_start()` is called when the validation was done as the first part of the commit, and validation succeeded.

If the underlying database is a real database with real support for transactions, it is a very good idea to start such a native transaction in the call to `write_start()`. If that is not the case the `libconfd.so` library provides support which makes it possible to accumulate the write operations without actually writing them.

In this example we save the database to a file for persistence

Example 7.7. save() utility function

```
static int save(char *filename) {
    FILE *fp;
    int i;
    if ((fp = fopen(filename, "w")) == NULL)
```

```
        return CONFD_ERR;
    for (i=0; i < num_servers; i++) {
        fprintf(fp, "%s %s %d\n",
                running_db[i].name,
                inet_ntoa(running_db[i].ip),
                running_db[i].port);
    }
    fclose(fp);
    return CONFD_OK;
}
```

We instantiate all the transaction callbacks and do the appropriate thing in each callback. Since the database is just a simple array, the variable `running_db`, we choose to let the library `libconfd.so` accumulate the individual write operations by returning `CONFD_ACCUMULATE` from the write callbacks `set_elem()`, `create()` and `remove()`. The data will be copied into data structures in the library.

The purpose of doing this is that we do not want to explicitly write into our local data structures in the write routines - rather we wish to delay this and perform the actual write operations in the `prepare()` callback.

Example 7.8. write callbacks using accumulate

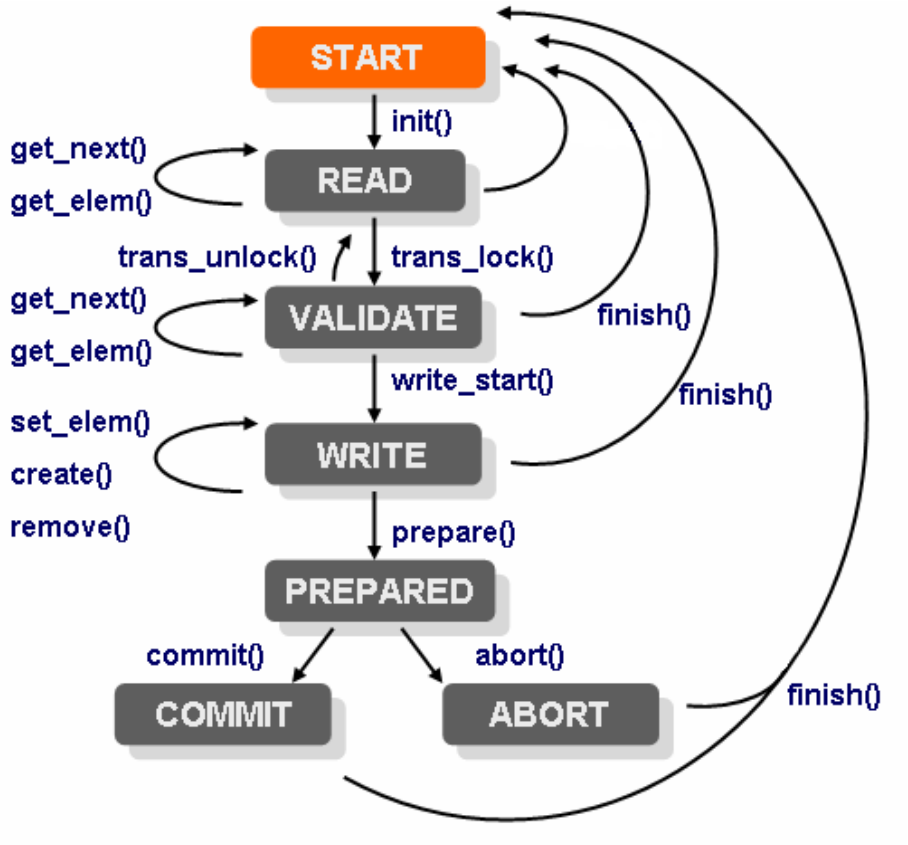
```
static int set_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    confd_value_t *newval)
{
    return CONFD_ACCUMULATE;
}
static int create(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    return CONFD_ACCUMULATE;
}

static int doremove(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    return CONFD_ACCUMULATE;
}
```

We are thus not doing anything at all in the write callbacks, except returning the value `CONFD_ACCUMULATE`. Note that this will store a complete copy of the keypath and also of the new value if the operation is `set_elem()`.

All the operations will be copied and kept in a linked list in the transaction context (struct confd_trans_ctx). In the `PREPARED` state we will loop through all the operations and perform them.

Remember the reason for implementing the two-phase commit protocol. There may be multiple `daemons` connected to ConfD and a series of write operations, i.e a transaction may span several `daemons`. ConfD ensures that e.g. a **commit** from the CLI is either written in all of the connected `daemons` or none - thus ensuring a consistent database.

Recall the picture depicting the state transitions:



ConfD transaction state machine

The most complicated callback is `prepare()`:

Example 7.9. prepare() callback using the accumulated write ops

```
static int t_prepare(struct confd_trans_ctx *tctx)
{
    struct server *s;
    struct confd_tr_item *item = tctx->accumulated;
    while (item) {
        confd_hkeypath_t *keypath = item->hkp;
        confd_value_t *leaf = &(keypath->v[0][0]);
        switch(item->op) {
        case C_SET_ELEM:
            s = find_server(&(keypath->v[1][0]));
            if (s == NULL)
                break;
            switch (CONFD_GET_XMLTAG(leaf)) {
            case smp_ip:
                s->ip = CONFD_GET_IPV4(item->val);
                break;
            case smp_port:
                s->port = CONFD_GET_INT32(item->val);
                break;

            }
            break;
        case C_CREATE:
```

```
            add_server((char *)CONFD_GET_BUFPTR(leaf));
            break;
        case C_REMOVE:
            remove_server(leaf);
            break;
        default:
            return CONFD_ERR;
        }
        item = item->next;
    }
    return save("running.prep");
}
```

The above code loops through all the struct confd_tr_item structs accumulated by the library in the `accumulated` field for the transaction context.

The accumulated write structs are defined as:

```
enum confd_tr_op {
    C_SET_ELEM = 1,
    C_CREATE= 2,
    C_REMOVE = 3,
    C_SET_CASE = 4,
    C_SET_ATTR = 5,
    C_MOVE_AFTER = 6
};
```

```
struct confd_tr_item {
    char *callpoint;
    enum confd_tr_op op;
    confd_hkeypath_t *hkp;
    confd_value_t *val;
    confd_value_t *choice;  /* only for set_case */
    u_int32_t attr;         /* only for set_attr */
    struct confd_tr_item *next;
};
```

If we had a real native database with real transaction support, we wouldn't have used the accumulation feature of the library at all - rather we would have started a native transaction in the `write_start()` callback.

Our example database is just an array and a file; thus we use the accumulation feature of the library.

In the `prepare()` callback we finally save the database to a file called `running.prep` - thus preparing to commit the changes we have made.

The corresponding `abort()` and `commit()` callbacks are easy:

Example 7.10. commit() and abort()

```
static int t_commit(struct confd_trans_ctx *tctx)
{
    if (rename("running.prep", "running.DB") == 0)
        return CONFD_OK;
    else
        return CONFD_ERR;
}

static int t_abort(struct confd_trans_ctx *tctx)
```

```
{
    restore("running.DB");
    unlink("running.prep");
    return CONFD_OK;
}
```

The `restore()` reads a file and initializes the database (our array) from that file:

Example 7.11. Code to restore our array from a file

```
static int restore(char *filename)
{
    char buf[BUFSIZ];
    FILE *fp;

    if ((fp = fopen(filename, "r")) == NULL)
        return CONFD_ERR;
    num_servers = 0;
    while (fgets(&buf[0], BUFSIZ, fp) != NULL) {
        char *name, *ip, *port;
        if ((name = strtok(buf, " \t\r\n")) != NULL &&
            ((ip = strtok(NULL, " \t\r\n")) != NULL) &&
            ((port = strtok(NULL, " \t\r\n")) != NULL)) {
            printf("Loaded %s\n", name);
            new_server(name, ip, port);
        }
    }
    return CONFD_OK;
}
```

7.8. Writable operational data

Writable operational data is indicated in the YANG model as `config false` marked with `tailf:writable true`. This is typically used when an SNMP MIB has data that models an operation, like "reboot". For other interfaces than SNMP, such an operation should be modeled as an rpc or action.

Writable operational data must be implemented by callback functions, just like external configuration data, as described in Section 7.7, “External configuration data with transactions”. When a transaction is started for operational data, the `dbname` field in `struct confd_trans_ctx` is `CONFD_OPERATIONAL`.

7.9. Supporting candidate commit

The NETCONF protocol has as one of its major features the concept of candidate commit with a timeout. The manager manipulates the candidate configuration and finally commits the candidate. This means that the candidate configuration is copied into the `running` data base and thus is active.

If the `commit` operation is accompanied by a timeout then the semantics is that if the application has not received a confirming commit before the timeout, the previous running configuration should be copied back into `running`. The idea here is that if a configuration is somehow bad, an automatic rollback will occur.

There are several different usage scenarios whereby this feature is supported with ConfD.

- The by far easiest case is when the database is kept in the ConfD built-in XML database, CDB. When that is the case, candidate commit is supported directly by ConfD natively.

- The next case is when the candidate configuration is managed by ConfD but the `running` configuration is kept outside ConfD. This is described here. The application needs to register three `checkpoint` callbacks in the database callback struct confd_db_cbs by means of the API call `confd_register_db_cb()`.
- The final case is when both the `running` and the `candidate` configuration are kept entirely outside of ConfD. Remember the ConfD transactions that get executed. When a new transaction is started, one of the fields in the transaction context, the `dbname` field indicates which database the transaction is started for.

  If ConfD owns the `candidate`, no transactions will ever be created towards the `candidate`. If the application owns both `running` and the `candidate` (as configured in `confd.conf`) then transaction may be directed towards either `running` or `candidate`.

  In the case where the candidate is owned by the application, the application needs to register six candidate callbacks in the database callback struct struct confd_db_cbs by means of the API call `confd_register_db_cb()`. This mode of operations only make sense if the external database can truly support the candidate callbacks. If that is not the case it i better to let ConfD manage the candidate.

In this section we provide an example where ConfD owns the candidate datastore. The application needs to register the following callbacks.

1. `add_checkpoint_running()` - This callback must create a checkpoint of the current running configuration and store it in non-volatile memory. When the system restarts, it is the responsibility of the external application to check if there is a checkpoint available, and use the checkpoint instead of running.
2. `del_checkpoint_running()` - This function must delete a checkpoint created by `add_checkpoint_running()`. It is called by ConfD when a confirming commit is received.
3. `activate_checkpoint_running()` - This function should rollback running to the checkpoint created by `add_checkpoint_running()`. It is called by ConfD when the timer expires or if the user session expires. There can be at most one checkpoint live at a time.

Using our previous `save()` and `restore()` functions the implementation of the checkpoint callbacks becomes very simple.

Example 7.12. checkpoint db callbacks

```
add_checkpoint_running(struct confd_db_ctx *db)
{
    return save("running.checkpoint");
}

del_checkpoint_running(struct confd_db_ctx *db)
{
    unlink("running.checkpoint");
    return CONFD_OK;
}

activate_checkpoint_running(struct confd_db_ctx *db)
{
    return restore("running.checkpoint");
}
```

Two things remain to be done. First we need to register the checkpoint callbacks. Second we need to look for the existence of a saved checkpoint when we initialize our database and if it exists, `running` should be initialized from the checkpoint instead. Thus:

```
/* global variable */
static struct confd_db_cbs dbcbs;

...

int main()
{
     ...

     if ((restore("running.checkpoint")) != CONFD_OK)
         restore("running.DB");



    dbcbs.add_checkpoint_running = add_checkpoint_running;
    dbcbs.del_checkpoint_running = del_checkpoint_running;
    dbcbs.activate_checkpoint_running = activate_checkpoint_running;

    /* register the callbacks */
    confd_register_db_cb(dctx, &dbcbs);
    confd_register_done(dctx);
```

If the underlying database is a real database we would install database checkpoints instead of copying entire files back and forth.

If we choose to implement the checkpoint callbacks as above, we must obviously also configure ConfD accordingly. The relevant sections in `confd.conf` from the `datastores` section are:

```
<candidate>
  <enabled>true</enabled>
  <implementation>confd</implementation>
</candidate>
```

And from the NETCONF section:

```
<capabilities>
  <candidate>
    <enabled>true</enabled>
  </candidate>

  <confirmed-commit>
    <enabled>true</enabled>
  </confirmed-commit>
</capabilities>
```

Finally, if we implement the database outside ConfD we may optionally choose to implement the `lock()` and `unlock()` callbacks. This is only interesting if there exists additional locking mechanisms towards the database - such as an external CLI which can lock the database, of if the external database owns the candidate.

7.10. Discussion - CDB versus external DB

In this section we discuss some of the requirements that an external database must be able to fulfill in order for ConfD to work properly. The reasons for choosing an external database as opposed to CDB may vary between projects. Some projects already have a database and the managed object code is already tightly coupled to that database. Other projects may feel that the underlying database must have characteristics that CDB doesn't have. It is certainly the case that CDB is not the best choice for, for example distributed

replication of large amounts of state data. CDB is not a check-pointing database for application state replication.

The first and most important requirement ConfD has on an external database is that it can execute transactions. The transaction manager inside ConfD will collect all data for a transaction and once the data has been validated, it will send the data as a series of write operations to the data provider. It is the responsibility of the database to execute this series of write operations atomically. Either they all get written or none. External databases that do not support transactions can still be used of course, but that then comes with the possibility of getting a corrupt configuration. Corruption will occur if:

1. Another data provider rejects the transaction - in this case ConfD will tell all data providers to abort. If there are no other data providers than the external database - this cannot happen.

2. ConfD dies while sending the write operations to the data provider, alternatively the network connectivity between ConfD and the data provider breaks. If this happens, the data provider never gets the whole transaction. One way of partially addressing this problem may be to make use of CONFD_ACCUMULATE feature whereby all writes are accumulated inside the library. That way the data provider at least can be certain that it has the entire transaction prior to starting its own write session.

Furthermore, CDB has two important features, schema upgrade and subscriptions. An external database must at least address this functionality.

Schema upgrade. When the YANG data model files are changed, CDB has the old schema - and its associated data - stored. On upgrade, CDB transforms all the old data so that it adheres to the new schema. If CDB is not used, the equivalent functionality must be performed by the external database.

Subscriptions - when the configuration is changed - the applications, the consumers of configuration data, must somehow be notified of the configuration changes. If CDB is not used, this is now the task of the external database.

Finally, if an external database is used, we must provide a mapping in the code of the data provider between ConfD keypaths and values to entries in the external database. For example, if we use a simple key/value database it's possible to write general code that works for all possible keypaths. The key is a confd_hkeypath_t and the value is obviously a confd_value_t. The only problem is how to handle `create()` and `delete()` operations for a key/value database. In the case of a delete operation, all children must also be deleted. It is easy to find the children since the schema is loaded in a data provider (through `confd_load_schemas()`) and a key/value data provider would then have to follow the schema, and delete all children.

Chapter 8. Configuration Meta-Data

8.1. Introduction to Configuration Meta-Data

In ConfD, meta-data can be associated with configuration data nodes. The meta-data is stored as `attributes` on data nodes in the configuration datastore. Having meta-data is optional, and requires support from the datastore implementation. CDB (see Chapter 5, *CDB - The ConfD XML Database* fully supports meta-data attributes, but if an external data provider (see Chapter 7, *The external database API* is used for configuration data, it needs to explicitly support meta-data attributes.

There are three meta-data attributes in ConfD, `annotation`, `tag`, and `inactive`. Each of these is discussed in the following sections.

To enable meta-data attributes, `/confdConfig/enableAttributes` in `confd.conf` (see confd.conf(5)) must be set to `true`.

8.2. Meta-Data: `annotation`

Any configuration data node can have at most one `annotation` attribute. An annotation is an arbitrary string which acts a comment for the node.

In the CLI, an annotation is set with the `annotate` command, and displayed as a comment. See Section 16.14, “Annotations and tags” for details.

```
admin@host% annotate interface eth0 "mgmt interface"

admin@host% show interface
/* mgmt interface */
interface eth0 {
  ...
}
```

In NETCONF, an annotation is created and display as an XML attribute, see Section 15.15, “Meta-data in Attributes” for details.

Annotations are not visible in the CDB API for CDB subscribers.

8.3. Meta-Data: `tag`

Any configuration data node can have a set of `tags` associated with it. Tags are set by the user for data organization and filtering purposes.

In the CLI, tags are administered with the **tag** command, and displayed as a comment with special syntax. It is also possible to filter the configuration based on how it is tagged. See Section 16.14, “Annotations and tags” for details.

In NETCONF, a tag is created and display as an XML attribute, see Section 15.15, “Meta-data in Attributes” for details. Standard XPath filtering can be used to filter the configuration based on how it is tagged.

Tags are not visible in the CDB API for CDB subscribers.

8.4. Meta-Data: `inactive`

Any existing, deletable data node can be marked as `inactive`. This has the same effect as deleting the node, except that it is still kept in the configuration data store, marked as being inactive.

To enable support for inactive nodes, `/confdConfig/enableInactive` in `confd.conf` (see confd.conf(5)) must be set to `true`. All configuration data providers must support the `inactive` attribute.

In the CLI, the command **deactivate** makes a node inactive, and the command **activate** activates an inactive node. See See Section 16.15, “Activate and Deactivate” for details.

In NETCONF, a separate capability is used by the server to announce that it supports inactive nodes. Clients must use special parameters to tell the server that they understand the inactive attribute. See Section 15.12, “Inactive Capability” for details.

Inactive nodes are not visible in the CDB API for CDB subscribers. If a node is inactivated, a CDB subscriber will see the node as being deleted, and when it is activated, the CDB subscriber will see it as being created.

Inactive nodes are not visible in validation code (see Chapter 9, *Semantic validation*). Validation constraints defined in the data model (e.g. `max-elements` or `must`) do not take inactive nodes into account.

Since data providers must support the `inactive` attribute, all hooks and transforms (see Chapter 10, *Transformations, Hooks, Hidden Data and Symlinks*) will see the inactive nodes being marked as inactive, and must be explicitly coded to handle this attribute.

```
admin@host% deactivate interface eth0

admin@host% show interface
inactive: interface eth-0 {
  ...
}
```

Chapter 9. Semantic validation

9.1. Why Do We Need to Validate

ConfD stores device configuration data. Some 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 correct.

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

ConfD has support for several different levels of validation. We have:

- Syntactic validation - this means that the configuration data - viewed as an XML document - must adhere to the YANG model.
- Integrity constraints. Certain configuration leaves may only have values within specified ranges.
- YANG must statements use XPath expressions that can be used to constrain values. This is a very powerful mechanism whereby it's 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.
- Explicit validation logic where user code gets to read and analyze the configuration prior to commit.

9.2. Syntactic Validation in YANG models

A YANG model is a schema. It has a number of constructs the define the structure of the model as a whole as well as type constrains on individual leaves.

Structure enforcing statements include constructs like container `presence` statements, leaf `mandatory` statements, leaf-list min and max elements statements and list min and max elements statements.

Each leaf has a either a built-in primitive type, e.g. integer, string, boolean etc, or a derived type e.g. a union, enumeration, boundary restriction, or regular expression pattern

The ConfD CLI and Web UI use this information to guide the operator what is possible to configure.

9.3. Integrity Constraints in YANG Models

Before going for semantic validation we should also make sure that our need for validation can not be satisfied by any of the *integrity constraint* constructs available in the YANG model modeling language:

`min-elements` and `max-elements`	Specifies how many instances may exist in the configuration data store. Both YANG list statements and leaf-list statements can be constrained by a min and/or a max.
`key`	Specifies that the leaf is used as a key for a multi-instance object. An object can have multiple keys.
`unique`	Specifies that the leaf's value must be unique across all instances.

`leafref` The leafref type is used to reference a particular leaf instance in the data tree. Its value is constrained to be the same as the value of an existing leaf.

Read more about integrity constraints in the Chapter 3, *The YANG Data Modeling Language* chapter as well in http://www.ietf.org/rfc/rfc6020.txt.

9.4. The YANG must Statement

Using XPath expressions it is possible to express constraints on values in virtually any way. XPath is complicated and requires a bit of work to learn. It is well worth the effort though, since writing declarative constraints on the data model - in the data model - is better than writing C code that executes outside of ConfD.

9.5. Validation Logic

Semantic validation in ConfD extends the validation functionality to allow for verification of constraints that can not be expressed by the above constructs. It is important to realize that the basic concept is the same, though. The task of semantic validation is to make sure that the new configuration satisfies some set of logical constraints before it is allowed to be committed.

With a command centric view of configuration, validation may be thought of as checking the validity of operator actions vis-a-vis existing configuration. This tends to lead to complex and error prone code, since there will often be a large number of combinations of actions and configuration values that need to be checked, and some "corner cases" can easily be overlooked. Furthermore covering multiple user interfaces, such as NETCONF and Web UI in addition to CLI, with the same validation code will be an almost impossible task.

In contrast, ConfD's data model centric validation concept, i.e. checking the validity of the configuration that will be the result of those actions, allows for clear and concise validation code that rejects an invalid configuration regardless of which commands or other operations that were used to (attempt to) create it.

Attempting to validate the operations instead of the resulting configuration can also lead to problems with loading config backups or doing rollbacks. The old configuration that should be applied as a result of such actions is obviously valid (as long as the logical constraints have not changed), but validation logic that rejects specific changes to the configuration may still result in that configuration being rejected.

An additional benefit of the "logical constraints" approach to validation is the possibility of "off-line" validation - i.e. a complete configuration can be loaded onto a device that is not in production use, and the validation code can give its verdict about its validity, even though the sequence of operations that would lead to this configuration may not even be known.

Since ConfD provides access to the complete new configuration inside a transaction for the purpose of semantic validation, it is generally straightforward to implement constraints that are expressed in terms of relations between configuration nodes that must hold true for any valid configuration. Identifying and formulating those constraints is thus the first thing we must do when implementing semantic validation.

There is however one case where using the validation functionality to check operator actions can be useful, namely when we want to warn the operator of undesirable consequences - e.g. "If you change this value, the system will reboot". This is not validation in the sense of verifying correctness, since the new configuration will be valid too and should not be rejected - but the validation code is still the appropriate place to implement such functionality. To specifically inspect changes from the current configuration to the new configuration, the MAAPI API provides functions to iterate over all or a subset of the changes, or if CDB is used, current configuration values can be read via the CDB API.

9.6. Validation Points

In a manner similar to how we use `callpoints` to register callback functions which read and write data in external databases, we use named *validation points* to define which code is responsible for the validation of different parts of the configuration. However unlike callpoints, validation points do not form a hierarchy where they "take over" responsibility from validation points higher up in the XML tree.

The validation code can reject the data, accept it, or accept it with a warning. If a warning is produced, it will be displayed for interactive users (e.g. through the CLI or Web UI). The user may choose to abort or continue to commit the transaction.

Validation callbacks are typically assigned to individual leafs or containers in the YANG model, but this is mostly a matter of organization and modularization of the validation code. In some cases it may even be feasible to use a single validation callback, e.g. on the top level node of the configuration. In such a case, this callback is responsible for the validation of all values and their relationships throughout the configuration.

A validation callback is only invoked if its validation point is for an element that exists in the new configuration. This may be surprising, but it is a logical consequence of the "validate the configuration, not the operations" concept of ConfD validation - we can not be asked to validate something that does not exist. Since it sometimes leads to the question "How can I prevent deletion of an element?", an example may be useful:

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

Here we can configure a notification protocol, and optionally the address of an SMTP server - we want it to be optional, since it is not needed unless the notification protocol is SMTP. However if protocol SMTP and a server has been configured, the SMTP server element must not be deleted. But if we assign a validation point to that element, the callback will not get invoked on deletion, since the element does not exist in the new configuration!

The solution is in the previous section - we must identify and formulate the logical constraint. In this case it is "If notification protocol SMTP is configured, an SMTP server must also be configured".

The easiest way to enforce this is through an XPath expression as in:

```
container notification {
   leaf protocol {
     must ". != 'SMTP' or ../smtp-server" {
       error-message "Must specify smtp-server";
     }
```

```
    type enumeration {
    ......
```

Alternatively, if we us C code logic to achieve the same thing to make sure that our callback is invoked, we assign the validation point to the "protocol" element. The validation callback will then get the value of this element as a parameter, and the implementation of the constraint check will just be:

```
if (CONFD_GET_ENUM_VALUE(newval) == nsprefix_SMTP &&
    maapi_exists(maapi_socket, tctx->thandle, "/notification/smtp-server") != 1) {
    confd_trans_seterr(tctx, "SMTP server must be configured");
    return CONFD_ERR;
}
```

See the examples below for the API details. Note well that since it is based on the logical constraint, this single expression also covers the other required case of "operational validation", i.e. setting of the notification protocol to SMTP - it will be rejected unless an SMTP server has been configured.

Note

Validation will always fail if no code is registered under a validation point that would otherwise have had its callback invoked during validation.

9.7. Validating Data in C

Next we describe how to connect user defined C code to the validation process. We start off with a really simple YANG model.

```
module mtest {
  namespace "http://tail-f.com/ns/example/mtest";
  prefix mtest;
  container mtest {
    leaf a_number {
      type int64;
      default 42;
    }
    leaf b_number {
      type int64;
      default 7;
    }
  }
}
```

We wish to ensure that the integer value */mtest/a_number* is bigger than */mtest/b_number*. We use a YANG model annotation file to specify the validation point. This is a good technique to use if wish to keep out data models clean of any Tail-f extensions.

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

  import mtest {
    prefix m;
  }
```

```
  import tailf-common {
    prefix tailf;
  }

  tailf:annotate "/m:mtest/m:a_number" {
    tailf:validate vp1;
  }
}
```

We define a validation point on */mtest/a_number* called `vp1`. This instructs ConfD that whenever ConfD needs to validate the XML data element associated with */mtest/a_number*, ConfD should call an external process which has registered itself under the validation point `vp1` using the `libconfd` interface. If no process is registered, the validation will fail and it will not be possible to commit any changes.

We continue with the necessary C code to implement the relevant parts of the external process. The complete code for this can be found in `examples.confd/validate/c` in the ConfD examples collection. The validation code will be called during the validation phase of a ConfD transaction, i.e. before any write operations have been performed. The C code must install three different callback functions.

- `init()` - This callback will be invoked for any transaction where one of our validate() callbacks is invoked. The purpose of the callback is to initialize data structures and sockets for the remainder of the transaction. In particular it must indicate to the library which socket should be used for this transaction.

  Another task to perform in the `init()` callback is to attach a MAAPI socket to this transaction. While validating the individual values, we use MAAPI to possibly read other XML data elements from the same transaction we are validating. The function `maapi_attach()` is used to attach our MAAPI socket to the running transaction.

- `validate()` - We may have several validation points. We must install one callback for each defined validation point. The callback will be automatically called by ConfD during the actual validation phase. The callback must return `CONFD_OK` if the validation succeeds, `CONFD_ERR` on validation failure, or `CONFD_VALIDATION_WARN` to accept with a warning.

- `stop()` - This callback will be invoked when the transaction finishes, if the `init()` callback was invoked. It will be called regardless of the outcome of the transaction.

The `init()` and `stop()` callbacks are installed through the API call `confd_register_trans_validate_cb()` and the individual validation point callbacks are installed through consecutive calls to `confd_register_valpoint_cb()` for each defined validation point.

We start by creating three sockets to ConfD. We need two sockets for the callback machinery and one MAAPI socket.

```
           #include "confd_lib.h"
#include "confd_dp.h"
#include "confd_maapi.h"

/* include generated ns file */
#include "mtest.h"

int debuglevel = CONFD_DEBUG;

static int ctlsock;
static int workersock;
static int maapi_socket;
```

```
static struct confd_daemon_ctx *dctx;


    confd_init("more_a_than_b", stderr, debuglevel);

    if ((dctx = confd_init_daemon("mydaemon")) == NULL)
        confd_fatal("Failed to initialize confd\n");

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

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

    OK(confd_load_schemas((struct sockaddr*)&addr, sizeof(struct sockaddr_in)));

    /* Create the first control socket, all requests to */
    /* create new transactions arrive here */
    if (confd_connect(dctx, ctlsock, CONTROL_SOCKET,
                      (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0) {
        confd_fatal("Failed to confd_connect() to confd \n");
    }

    /* Also establish a workersocket, this is the most simple */
    /* case where we have just one ctlsock and one workersock */
    if ((workersock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open workersocket\n");
    if (confd_connect(dctx, workersock, WORKER_SOCKET,(struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

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

    if (maapi_connect(*maapi_sock, (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");
```

The above code connects three times. We need the control socket and the worker socket for the C callbacks. This works precisely the same way as when C callbacks are installed for the external data provider API. Thus the request from ConfD to invoke the `init()` callback will arrive on the control socket whereas the subsequent requests to invoke the individual `validate()` callbacks as well as the finishing request to invoke `stop()` will arrive on the designated worker socket.

The MAAPI socket will be used to attach the running transaction to a MAAPI socket.

All three sockets are connected to the same port number.

Next step is to continue with the installation of the callbacks:

```
            struct confd_trans_validate_cbs vcb;
struct confd_valpoint_cb valp1;

static void OK(int rval)
```

```
{
    if (rval != CONFD_OK) {
        fprintf(stderr, "more_a_than_b.c: error not CONFD_OK: %d : %s \n",
                confd_errno, confd_lasterr());
        abort();
    }
}

                vcb.init = init_validation;
    vcb.stop = stop_validation;
    confd_register_trans_validate_cb(dctx, &vcb);

    valp1.validate = validate;
    strcpy(valp1.valpoint, "vp1");
    OK(confd_register_valpoint_cb(dctx, &valp1));

    OK(confd_register_done(dctx));
```

Note the call to `confd_register_done()` after the callback registrations - this is required, to tell ConfD that we have completed our registrations. The actual callbacks look like:

```
            static int init_validation(struct confd_trans_ctx *tctx)
{
    OK(maapi_attach(maapi_socket, mtest__ns, tctx));
    confd_trans_set_fd(tctx, workersock);
    return CONFD_OK;
}

static int stop_validation(struct confd_trans_ctx *tctx)
{
    OK(maapi_detach(maapi_socket, tctx));
    return CONFD_OK;
}

static int validate(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    confd_value_t *newval)
{
    int64_t b_val;
    int64_t a_val;
    int64_t newval_a;

    /* we validate that a_number > b_number  */

    newval_a = CONFD_GET_INT64(newval);

    /* this switch is not necessary in this case; we know that we're
       called for a_number only.  the switch is useful when the same
       code is used to validate multiple objects. */
    switch (CONFD_GET_XMLTAG(&(keypath->v[0][0]))) {
    case mtest_a_number:
        OK(maapi_get_int64_elem(maapi_socket, tctx->thandle, &b_val,
                                "/mtest/b_number"));
        OK(maapi_get_int64_elem(maapi_socket, tctx->thandle, &a_val,
                                "/mtest/a_number"));
        /* just an assertion to show that newval == /mtest/a_number */
        /* in this transaction */
```

```
        assert(CONFD_GET_INT64(newval) == a_val);
        if (newval_a == 88) {
            /* This is how we get to interact with the CLI/webui */
            confd_trans_seterr(tctx, "Dangerous value: 88");
            return CONFD_VALIDATION_WARN;
        }
        else if (newval_a >  b_val) {
            return CONFD_OK;
        }
        else {
            confd_trans_seterr(tctx, "a_number is <= b_number ");
            return CONFD_ERR;
        }
        break;

    default: {
        char ebuf[BUFSIZ];
        sprintf(ebuf, "Unknown tag %d",
                CONFD_GET_XMLTAG(&(keypath->v[0][0])));
        confd_trans_seterr(tctx, ebuf);
        return CONFD_ERR;
    } /* default case */
    } /* switch */
}
```

The `switch` on the keypath exemplifies that we really get the keypath populated with the path leading to the textual element being validated. We can thus have the same validation point validate different XML data elements.

The `init` callback attaches to MAAPI and the global variable `maapi_socket` is used to read data from the transaction. All MAAPI functions use a "transaction handle"; this handle is available inside the instantiated `struct confd_trans_ctx *tctx` structure.

Finally we have a poll loop where we dispatch requests to invoke C callbacks on the control socket and the worker socket.

```
                while (1) {
        struct pollfd set[2];
        int ret;

        set[0].fd = ctlsock;
        set[0].events = POLLIN;
        set[0].revents = 0;

        set[1].fd = workersock;
        set[1].events = POLLIN;
        set[1].revents = 0;

        if (poll(&set[0], 2, -1) < 0) {
            perror("Poll failed:");
            continue;
        }

        if (set[0].revents & POLLIN) {
            if ((ret = confd_fd_ready(dctx, ctlsock)) == CONFD_EOF) {
```

```
                confd_fatal("Control socket closed\n");
            } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
                confd_fatal("Error on control socket request\n");
            }
        }
        if (set[1].revents & POLLIN) {
            if ((ret = confd_fd_ready(dctx, workersock)) == CONFD_EOF) {
                confd_fatal("Worker socket closed\n");
            } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
                confd_fatal("Error on worker socket request\n");
            }
        }
    }
```

In the above example we also showed how to issue a warning as opposed to a validation failure. Hadn't it been for that, it would have been considerably easier to express the same validation as an XPath expression. Thus we attach a `must` statement to the `mtest` container as:

```
container mtest {
  must "a_number > b_number" {
    error-message "a_number is <= b_number";
  }
```

9.8. Validation Points and CDB

When CDB first starts or upgrades the database it creates a special transaction which, when committed, will invoke validation. An external validation point (written e.g. in C) has to be registered before these transactions are committed, otherwise starting ConfD will fail. Starting ConfD and external applications in a synchronized way is accomplished using ConfD start phases (see the Advanced Topics chapter). To avoid this extra complexity use the `--ignore-initial-validation` option when starting ConfD (useful during development).

In a validation point it might be desirable to access CDB to validate a value against the old value of the parameter (or some other parameter) in the configuration. Using the normal cdb calls this works fine in the normal case, but when CDB is initializing there are no old values. The `cdb_get_phase()` call can be used to check for this case, (see the confd_lib_cdb(3) manual page for details).

Note

If a CDB session is used throughout the validation phase (i.e. the session is not ended until the `stop()` callback invocation), we must start it without a read lock, i.e. using `cdb_start_session2()` with `flags` = 0. It is safe to do that in this particular case, since the transaction lock prevents changes to CDB during validation.

9.9. Dependencies - Why Does Validation Points Get Called

In general, validation code for a particular element in the configuration may read any other part of the configuration, and accept or reject the configuration based on that. I.e. the outcome of the validation may

actually depend on other configuration elements than the one the validation point is assigned to - and for correct operation, the validation code must be executed when any element it depends on has been modified. As ConfD can not make any assumptions about these dependencies, it takes the safe default of always invoking all callbacks (for existing elements) on every configuration change.

It is possible to declare these dependencies explicitly in the YANG model. This can be a significant optimization, but it is strictly an optimization, i.e. a validation callback implementing a logical constraint verification will always return the same result for a given configuration, it doesn't matter if it is invoked unnecessarily due to lack of a dependency declaration in the YANG model. On the other hand an incorrect dependency declaration, that omits some dependency, can allow changes that lead to an invalid configuration. Thus if dependency declarations are used, it is critical that they are correct, and in particular that they are updated as needed if the validation logic of the callback is changed.

There can be multiple dependency declarations for a validation point. Each declaration consists of a `dependency` element specifying a configuration subtree that the validation code is dependent upon. If any element in any of the subtrees is modified, the validation callback is invoked. A subtree can be specified as an absolute path or as a relative path.

The relative path '.' is often used to declare that the validation code needs to be run whenever the current element or an element below it is modified. However note that per above, routinely specifying '.' as the *only* dependency for all validation points is a dangerous practice - if the validation logic actually depends on elements outside the subtree of the validation point, an invalid configuration may go undetected. Also, for a leaf element, having '.' as the only dependency is almost always wrong - if the validation really depends only on the leaf itself, it is likely that it could be expressed as a constraint in the YANG model instead of via a validation callback.

As described above, if a dependency is not declared, it defaults to a single dependency on the root of the configuration tree (/), which means that the validation code is executed when any configuration element is modified.

If dependencies are declared on a leaf element, an implicit dependency on the leaf itself is added.

As an example, consider the */mtest/a_number* validation above. The element `a_number` has validation code attached to it, and this code depends on element `b_number`. Thus, this code has to be executed whenever `a_number` or `b_number` is modified. To specify this, we can do:

```
leaf a_number {
  type int64;
  default 42;
  tailf:validate vp1 {
    tailf:dependency '../b_number';
  }
}
```

Here we specified the validation point with its `dependency` sub-element directly in the YANG model - it is of course possible to use an annotation file in this case too.

It is also possible, and recommended for performance reasons, to specify dependencies in `must` statements:

```
leaf a_number {
  type int64;
  default 42;
  must ". > ../b_number" {
```

```
    tailf:dependency '../b_number';
  }
}
```

The compiler gives a warning if a `must` statement lacks a `tailf:dependency` statement, and it cannot derive the dependency from the expression. The options `--fail-on-warnings` or `-E TAILF_MUST_NEED_DEPENDENCY` can be given to force this warning to be treated as an error.

9.10. Configuration Policies

Configuration policies is an optional mechanism by which the operator of a ConfD-based system can define its own custom validation rules. A configuration policy enforces custom validation rules on the configuration data. These rules assert that the user-defined conditions are always true in committed data. If a configuration change is done such that a policy rule would evaluate to false, the configuration change is rejected by the system.

As an example, an operator might define a configuration policy that bgp must never be disabled on a device, or define a policy that the MTU on SONET interfaces must be greater than 2048.

The data model for configuration policies is defined in `tailf-configuration-policy.yang`, in the directory `$CONFD_DIR/src/confd/configuration_policy/`. The YANG model contains a Also included in this directory is a pre-compiled .fxs file, and a `Makefile` that can be modified as necessary, for example to compile the fxs file with a *`--export`* parameter to **confdc**.

To enable this optional feature, put the `tailf-configuration-policy.fxs` in the load path to ConfD.

9.10.1. Example

These examples, and more, are available in `examples.confd/validate/configuration_policy` in the distribution.

As a first simple example, we define a policy that makes sure that BGP is always enabled on the box. The example assumes the following data model:

```
container protocols {
  container bgp {
    presence "enables bgp";
    // BGP config goes here...
  }
}
```

The following CLI commands, define the policy we need:

```
admin@host% configure

admin@host% set policy rule chk-bgp expr "/protocols/bgp"

admin@host% set policy rule chk-bgp error-message "bgp must be enabled"

admin@host% commit
Commit complete.
```

Now, let's try to disable bgp:

```
admin@host% delete protocols bgp

admin@host% commit
Aborted: bgp must be enabled
```

As another example, we define a policy that ensures that the MTU of SONET interfaces are greater than or equal than 2048:

```
admin@host> set autowizard false

admin@host> configure

admin@host% set policy rule chk-sonet-mtu

admin@host% edit policy rule chk-sonet-mtu

admin@host% set foreach "/interface[type='sonet']"

admin@host% set expr "mtu >= 2048"

admin@host% set error-message "Sonet interface {name} has MTU {mtu}, must be at least 2048"

admin@host% top

admin@host% commit
Commit complete.
[ok][2010-11-02 09:39:00]
```

This rule uses the `foreach` leaf. When ConfD evaluates this rule, it will first evaluate the `foreach` expression. This expression evaluates to a node set with all sonet interfaces. Then, foreach node in this node set, the `expr` expression is evaluated. If it evaluates to false, validation fails with the error message given.

The error message uses a special notation `{<xpath-expression>}`. Before the error message is printed, ConfD substitutes all XPath expressions within `{ }`, by converting the result to a string. In this example, there are two such XPath expressions `{name}` and `{mtu}`.

This is best shown in an example:

```
mbj@x15% set interface so-1/0 type sonet mtu 4096

mbj@x15% set interface so-1/1 type sonet mtu 4096

mbj@x15% set interface so-1/2 type sonet mtu 1024

mbj@x15% validate
Failed: Sonet interface so-1/2 has MTU 1024, must be at least 2048
```

Chapter 10. Transformations, Hooks, Hidden Data and Symlinks

10.1. Introduction

When building new variants of an old product, we often have a situation where we have large amounts of application code, which reads configuration data from some datastore and we do not want to make any changes to the application code, we merely wish to expose a different view of the same configuration data.

Another common situation is when we have application code which requires more configuration data than we wish to expose through the northbound management interfaces. The application reads and use a number of configuration items that do not make sense to expose through the different management interfaces.

In general, when the actual configuration data differs from what we wish to expose as management data, we can use a variety of different techniques in ConfD.

In this chapter we describe the following four different techniques that all are used to somehow show different views of the system.

- *Transforms* are used to show a portion of the system in a different way than the original data model.
- *Hooks* are used to execute user code whenever a part of the configuration is changed.
- *Hidden Data* is means to hide parts of the configuration.
- *Symlinks* are pointers in the data model effectively making one part of the data model appear at a different place.

Some or several of the above features are often the required trick when we wish to accomplish certain modifications of the data model. However, combining these features, can make the system complex. A clean YANG data model is easy to understand - whereas a system consisting of hidden transformations with symlinks here and there, can very easy become hard to understand. So - use these features with caution.

10.2. Transformation Control Flow

ConfD uses the data model, i.e. the YANG files to render all the northbound interfaces, the layout of the datamodel is exactly reflected in the CLI and the Web UI. Thus, unfortunately, we may sometimes be forced to manipulate the data model in order to have a desired look and feel in the CLI or the Web UI. In these cases, a transform may be required trick.

A transformation works as follows. We start out with the YANG model we wish to transform. We may add **tailf:export maapi** to that YANG module. This makes the YANG model invisible to all management agents except MAAPI. Although invisible, the YANG model is loaded into the system and regardless of whether the YANG model is populated through cdb or an external database it is fully operational and accessible through the MAAPI APIs as well as through the CDB APIs (assuming the YANG model is populated by CDB).

Following that, we write another YANG model which represents the management data we do wish to expose to the northbound management agents. This YANG model has a callpoint which uses the attribute `tailf:transform`. Finally we must write a program which acts as an external database, serving data for a callpoint. This program should use the MAAPI API to read and write the real YANG model data which is used by the managed objects.

10.3. An Example

Assume we have the following YANG model:

Example 10.1. full.yang

```
container full {
  leaf firstname {
    type string;
    default George;
  }
  leaf a_number {
    type int64;
    default 42;
  }
  leaf b_number {
    type int64;
    default 7;
  }
  container servers {
    list server {
      key name;
      max-elements 64;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:host;
        mandatory true;
      }
      leaf port {
        type inet:port-number;
        mandatory true;
      }
    }
  }
}
```

For some reason we think that this YANG model is way too complicated to expose through the management interfaces. We have also invested time and energy in various applications that read and use precisely this data and we do not want to change any of those applications.

What we want to do is to expose a YANG model which looks like:

Example 10.2. small.yang

```
container small {
  container servers {
    tailf:callpoint transcp {
        tailf:transform true;
    }
    list server {
      key name;
      max-elements 64;
      leaf name {
```

```
        type string;
      }
    }
  }
}
```

I.e. skip all the first toplevel elements, and skip the ip and the port. We write C code which derives both. Also note the transformation callpoint.

When ConfD needs to read and write data in the `small.yang` YANG model, it will use the callpoint `transcp`. I.e. it will invoke the installed callback functions for that callpoint. The main difference between a normal callpoint and a transformation callpoint is when write and when validation occurs. In a transformation callpoint the write operations occur before validation. This means that any data that is written through MAAPI in the actual transform, will also be validated.

Similar to callpoints for external data, we need a worker socket a control socket and registered callbacks. Our `main()` function together with some global variables would look like:

```
#include "full.h"  /* generated .h files */
#include "small.h"

static struct confd_daemon_ctx *dctx;
static int ctlsock;
static int workersock;
struct confd_trans_cbs tcb;
struct confd_data_cbs  data;
static int maapi_socket;


int main()
{

    struct in_addr in;
    struct sockaddr_in addr;

    confd_init("MYNAME", stderr, debuglevel);
    if ((dctx = confd_init_daemon("mydaemon")) == NULL)
        confd_fatal("Failed to initialize confd\n");

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

    inet_aton("127.0.0.1", &in);
    addr.sin_addr.s_addr = in.s_addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(CONFD_PORT);
    confd_load_schemas((struct sockaddr*)&addr,sizeof (struct sockaddr_in))l
    if (confd_connect(dctx, ctlsock, CONTROL_SOCKET, (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

    if ((workersock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        confd_fatal("Failed to open workersocket\n");
    if (confd_connect(dctx, workersock, WORKER_SOCKET,(struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");
```

```
    tcb.init = init_transformation;
    tcb.finish = stop_transformation;
    confd_register_trans_cb(dctx, &tcb);


    data.get_elem = get_elem;
    data.get_next = get_next;
    data.set_elem = set_elem;
    data.create   = create;
    data.remove   = dbremove;
    data.exists_optional = NULL;
    strcpy(data.callpoint, "transcp");

    if (confd_register_data_cb(dctx, &data) != CONFD_OK)
        confd_fatal("Failed to register data cb \n");

    if (confd_register_done(dctx) != CONFD_OK)
        confd_fatal("Failed to complete registration \n");

    setup_maapi_sock(&maapi_socket));

    .........
```

The above is precisely the same setup as when we register callbacks for an external database with the only exception that the callpoint uses `tailf:transform true`.

The code to establish the MAAPI socket is just a call to `maapi_connect()`.

The difference comes in the implementation of the data callbacks, with an external database, we have the data to deliver, in the case of a transformation callpoint, we don't have the data. The data resides inside ConfD and we can read and write that data *inside* the same transaction using `maapi_attach()`.

The initialization looks like:

```
static int init_transformation(struct confd_trans_ctx *tctx)
{
    maapi_attach(maapi_socket, full__ns, tctx);
    confd_trans_set_fd(tctx, workersock);
    return CONFD_OK;
}

static int stop_transformation(struct confd_trans_ctx *tctx)
{
    if (tctx->t_opaque != NULL) {
        struct maapi_cursor *mc = (struct maapi_cursor *)tctx->t_opaque;
        maapi_destroy_cursor(mc);
        free(tctx->t_opaque);
    }
    maapi_detach(maapi_socket, tctx);
    return CONFD_OK;
}
```

Whenever a transaction starts, we get called - as usual - in our `init()` callback and whenever the transaction terminates, regardless of the outcome of the transaction, we get called in our `finish()` callback. Here we attach to the executing transaction using `maapi_attach()` in the `init()` callback. We will

use the attached MAAPI socket with the right transaction handle in all our data processing callbacks. In the `finish()` callback we need to release any memory used for a MAAPI cursor, see `get_next()` below.

The `get_elem()` callback is interesting. The path we get queried with is `/small/servers/server{key}/name` which doesn't exist in the real database. What does exist as proper data though is the path `/full/servers/server{key}/name` and we use the MAAPI socket to read that value from the "hidden" YANG model.

```
static int get_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)

{

    confd_value_t v;
    confd_value_t *leaf = &(keypath->v[0][0]);
    confd_value_t *vp = &(keypath->v[1][0]);
    switch (CONFD_GET_XMLTAG(leaf)) {
    case small_name:
        if (maapi_get_elem(maapi_socket, tctx->thandle, &v,
                           "/full/servers/server{%x}/name",  vp) == CONFD_OK) {
            confd_data_reply_value(tctx, &v);
            free(CONFD_GET_BUFPTR(&v));
            return CONFD_OK;
        }
        else if (confd_errno == CONFD_ERR_NOEXISTS) {
            fprintf(stderr, "\nNOT FOUND \n");
            confd_data_reply_not_found(tctx);
            return CONFD_OK;
        }
        else {
            fprintf (stderr, "errno = %d\n", confd_errno);
            return CONFD_ERR;
        }
    default:
        return CONFD_ERR;
    }
}
```

It is important that we check `confd_errno` to distinguish between the real error cases and the case where the element doesn't exist. Also note how we format the path to the `maapi_get_elem()` call using the second element in the keypath. This will be the key since the path will be `/small/servers/server{key}/name`.

`set_elem()` will never be called since our MO *server* only contains a key and no other elements.

The callback `get_next()` is also interesting. Here we utilize a MAAPI cursor to iterate through the different "full" servers. Since the MAAPI cursor must exist across calls to `get_next()`, and we must be able to handle multiple transactions (from different user sessions) in parallel, we allocate the cursor dynamically and use the `t_opaque` element in the transaction context to keep track of it.

```
static int get_next(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    long next)
{
    struct maapi_cursor *mc;
```

```
    if (next == -1) {
        if (tctx->t_opaque == NULL) {
            /* allocate the cursor */
            mc = (struct maapi_cursor *)malloc(sizeof(struct maapi_cursor));
            tctx->t_opaque = mc;
        } else {
            /* re-use previously allocated cursor */
            mc = (struct maapi_cursor *)tctx->t_opaque;
            maapi_destroy_cursor(mc);
        }
        maapi_init_cursor(maapi_socket, tctx->thandle, mc,
                          "/full/servers/server");
    } else {
        mc = (struct maapi_cursor *)tctx->t_opaque;
    }
    maapi_get_next(mc);
    if (mc->n == 0) {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }
    confd_data_reply_next_key(tctx, &(mc->keys[0]), 1, 1);
    return CONFD_OK;
}
```

Finally we have the `delete()` and `create()` callbacks. The `delete()` callback is completely straightforward where we simply delete the same element from the hidden "full" YANG model. The `create()` callback needs to do a bit of work. The "full" YANG model contains two elements that are not part of the "small" YANG model. Thus when a manager creates a new element in the "small" YANG model, it is the responsibility of our code here to create the corresponding element in the "full" YANG model, but also to populate the additional two elements with values. If we fail to do that the commit will fail since values in the "full" YANG model are unset.

The code to delete and create:

```
static int dbremove(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)
{
    maapi_delete(maapi_socket, tctx->thandle,
                 "/full/servers/server{%x}", &(keypath->v[0][0]));
    return CONFD_OK;
}


static int create(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    /* this is where we have to do extra, we need to also populate */
    /* the ip and ports fields in full.cs */
    char buf[BUFSIZ];
    confd_value_t *key = &(keypath->v[0][0]);
    struct servent *srv;
    maapi_create(maapi_socket, tctx->thandle,
                 "/full/servers/server{%x}", key);
    maapi_set_elem2(maapi_socket, tctx->thandle,
                    "0.0.0.0",
                    "/full/servers/server{%x}/ip",key);
```

```
    /* NUL terminate string */
    memcpy(buf, CONFD_GET_BUFPTR(key), CONFD_GET_BUFSIZE(key));
    buf[CONFD_GET_BUFSIZE(key)] = 0;

    if ((srv = getservbyname(buf, NULL)) == NULL) {
        char tbuf[BUFSIZ];
        sprintf(tbuf, "Unknown service %s", buf);
        confd_trans_seterr(tctx, tbuf);
        return CONFD_ERR;
    }
    sprintf(buf, "%d", srv->s_port);
    maapi_set_elem2(maapi_socket, tctx->thandle, buf,
                    "/full/servers/server{%x}/port",key);
    return CONFD_OK;
}
```

10.4. AAA Transform

The ConfD AAA YANG model is a very good example of where we may wish to expose a different set of configuration items to the management stations than what exists in the AAA YANG model `tailf-aaa.yang`. The AAA system is described in Chapter 14, *The AAA infrastructure*.

The data in the AAA YANG model is used by ConfD itself and all that data including the fairly complicated authorization rules must be there for ConfD to read. We think that very few devices wish to expose e.g. the authorization rules from `tailf-aaa.yang` to end users. The solution to this is to use a transformation.

In the ConfD examples collection, we have an example which exposes a very simple AAA model. The simple AAA YANG model looks as:

Example 10.3. users.yang

```
module users {
  namespace "http://www.example.com/ns/users";
  prefix u;

  import tailf-common {
    prefix tailf;
  }

  typedef Role {
    type enumeration {
        enum admin;
        enum oper;
    }
  }
  typedef passwdStr {
    type tailf:md5-digest-string {
    }
  }
  container users {
    tailf:callpoint simple_aaa {
        tailf:transform true;
    }
    list user {
      key name;
      max-elements 64;
```

```
      leaf name {
        type string;
      }
      leaf password {
        type passwdStr;
        mandatory true;
      }
      leaf role {
        type Role;
        mandatory true;
      }
    }
  }
}
```

This YANG model just exposes a list of users. Each user has a password and an enum indicating the role of the user. There are only two static roles to choose from, `admin` and `oper`.

If we also have intimate knowledge of the datamodel we are using, it is possible to generate static authorization rules.

Let's take a look at the `get_elem()` callback. The task of this callback is to use MAAPI in order to populate the simple YANG model from above.

```
static int get_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)

{

    confd_value_t v;
    confd_value_t *leaf = &(keypath->v[0][0]);
    confd_value_t *vp = &(keypath->v[1][0]);
    switch (CONFD_GET_XMLTAG(leaf)) {
    case aaa_simple_name:
        if (maapi_get_elem(
                maapi_socket, tctx->thandle, &v,
                "/aaa/authentication/users/user{%x}/name",  vp) == CONFD_OK) {
            confd_data_reply_value(tctx, &v);
            free(CONFD_GET_BUFPTR(&v));
            return CONFD_OK;
        }
        else if (confd_errno == CONFD_ERR_NOEXISTS) {
            confd_data_reply_not_found(tctx);
            return CONFD_OK;
        }
        else {
            printf ("errno = %d\n", confd_errno);
            return CONFD_ERR;
        }
    case aaa_simple_password:
        if (maapi_get_elem(
                maapi_socket, tctx->thandle, &v,
                "/aaa/authentication/users/user{%x}/password", vp)==CONFD_OK) {
            confd_data_reply_value(tctx, &v);
            free(CONFD_GET_BUFPTR(&v));
            return CONFD_OK;
        }
        else if (confd_errno == CONFD_ERR_NOEXISTS) {
```

```
            confd_data_reply_not_found(tctx);
            return CONFD_OK;
        }
        else {
            fprintf (stderr, "errno = %d\n", confd_errno);
            return CONFD_ERR;
        }
    case aaa_simple_role: {
        int ret;
        char users[BUFSIZ];
        char user[256];
        memcpy(&user[0], CONFD_GET_BUFPTR(vp), CONFD_GET_BUFSIZE(vp));
        user[CONFD_GET_BUFSIZE(vp)] = 0;

        ret = maapi_get_str_elem(
            maapi_socket, tctx->thandle, users, BUFSIZ,
            "/aaa/authentication/groups/group{admin}/users");
        if (strstr(users, user) != NULL) {
            CONFD_SET_ENUM_VALUE(&v, aaa_simple_admin);
            confd_data_reply_value(tctx, &v);
            return CONFD_OK;
        }
        else {
            maapi_get_str_elem(
                maapi_socket, tctx->thandle, users, BUFSIZ,
                "/aaa/authentication/groups/group{oper}/users");
            if (strstr(users, user) != NULL) {
                CONFD_SET_ENUM_VALUE(&v, aaa_simple_oper);
                confd_data_reply_value(tctx, &v);
                return CONFD_OK;
            }
        }
        /* user not part of any group at all */
        confd_data_reply_not_found(tctx);
        return CONFD_OK;
    }
    default:
        confd_fatal("Unexpected switch tag %d\n",
                    CONFD_GET_XMLTAG(leaf));
    }
    return CONFD_ERR;
}
```

10.5. Other Use Cases for Transformations

We may also envision a use case where we wish to expose more data than is available in the YANG model. In this case the task of the transformation would be to aggregate the data and write into MAAPI.

Yet another use case for transformations would be when we wish to expose two variants of the same config, one for novices and one for experts. In this case we have the full YANG model with all the details exposed to experts and a simplified version which fills in many reasonable default values and possibly also derives data, exposed to novices. The YANG model for novices would then be populated by a transformation.

One common transformation is to logically move an entire subtree of some data model to some other place. For example, ConfD's AAA data model is named `/aaa`, but suppose we want to access it through `/system/advanced/aaa` instead. This can be done by using a transform as described above, or it can be done using a `symlink`, which essentially is a specialized, built-in transform.

Finally when we want to implement support for standard SNMP mibs while at the same time use a proper hierarchical high level data model for all other north bound interfaces we must use a transform on the SNMP data. To implement this we compile the mib into a YANG model document using the `smidump` tool. We must then also annotate the YANG model derived from the MIB and set a transformation point at the top. Thus when the SNMP agent tries to read data from the MIB (through the YANG model) our transformation C code gets invoked and we can then, over the maapi interface, read the right data from the high level data model and return that data to the SNMP agent.

10.6. Hooks

A hook is a function that is invoked within the transaction when an object is modified. The hook function has access to the transaction, so it can modify other objects in the transaction as necessary.

A hook is a way for the application to participate in a transaction. A hook is like a callpoint or a validation point only that the application gets to attach (using maapi) to the transaction and can write more data.

For example if we have an optional container containing a set of items, whenever the container is created, our hook gets called and the container can be populated with proper values. This effect is also achieved by letting the container elements have default values. The "default value" solution is compile time, whereas a populating the container through a hook is obviously runtime.

Hooks can also be used to attach some magic to individual elements. Say that we have a leaf:

```
leaf magic {
   type int32;
}
```

Whenever the *magic* leaf get set to, say *-1*, our hook code performs some other arbitrary write operations.

Thus the hook mechanism can be used to achieve a wide variety of effects.

A hook is implemented similar to a callpoint with the exception that only write callbacks need to be implemented. The write callbacks are `set_elem()`, `create()`, `remove()`, `set_case()`, `set_attr()`, and `move_after()`. However a hook only needs to implement the write callbacks that it actually needs for its own use - the others can be left unimplemented, indicated by setting them to NULL in the callback registration.

There are two types of hooks, *set hooks* and *transaction hooks*. The main difference between the two is when they are invoked. Set hooks are invoked directly as an object is modified, and transaction hooks are invoked in the two-phase commit phase. The ConfD transaction engine receives all the original write operations from one of the north bound agents. Once all write operations have been received, i.e. when a user for example types "commit" in the CLI, the transaction engine invokes the relevant transaction hooks. Once all transaction hooks are run the validation phase is entered, thus the write operations performed by the transaction hooks are also validated.

When set hooks make changes to the configuration, these changes are just like changes done directly by the user - they will be committed together with other changes, e.g. when the user types "commit" in the CLI. If such a commit operation cannot be completed, due to validation errors, the changes done by set hooks will remain in the change set until explicitly reverted.

Changes done by transaction hooks are different, in that ConfD keeps track of them, and rolls them back in case a commit operation cannot be completed. This makes it possible for the user to fix validation errors and attempt to commit again.

The hook is specified similar to callpoint if we have:

```
list dyn {
  key name;
  max-elements 64;
  tailf:callpoint foocp {
      tailf:transaction-hook subtree;
  }
  leaf name {
    type string;
  }
  leaf aval {
    type string;
    mandatory true;
  }
  leaf container {
    type empty;
  }
}
```

The statement: `tailf:transaction-hook subtree;` indicates that we wish to attach a hook to this part of the data model. A set hook uses the statement `tailf:set-hook` instead. Similar to a validator, the hook code will participate in the transaction by calling `maapi_attach()` and similar to a data provider, the hook code must register its callbacks through calls to `confd_register_trans_cb()` and `confd_register_data_cb()`. Only those of the possible write callbacks that are actually needed by the hook need to be registered as data callbacks. All other callbacks must be set to NULL.

For example our `set_elem()` callback could look like:

```
static int my_set_elem(struct confd_trans_ctx *tctx,
                       confd_hkeypath_t *keypath,
                       confd_value_t *newval)
{
    confd_value_t *tag = &(keypath->v[0][0]);
    confd_value_t *key = &(keypath->v[1][0]);

    if (confd_svcmp("donk", key) == 0) {
        maapi_setelem2(maapisock, tctx->thandle,"222", "/foo/bar");
        .....
```

So whenever some north bound agent assigns the value "donk" to `/dyn{key}/aval` for all values of `key` our code kicks in and additionally assigns a value to `/foo/bar`.

We can have three different kinds of hooks.

1. `subtree` - this assigns the hook code to all objects found below where the hook is defined. The value `"true"` is the same as `"subtree"`.

2. `object` - This is used when we wish to assign a hook to the manipulation of list entries. The hook reaches down to and including the list where it is defined. If there exists further lists further down in the tree they are not affected by the hook.

3. `node` - This is used when we wish to assign a hook an optional container and only that. It affects the container but non of its children.

In some cases a transaction hook may need to update the transaction in a way that really depends on the complete configuration, rather than on the changes done in the current transaction, making it dif-

ficult to implement the hook via the `create()`, `set_elem()`, etc callbacks. We can then use the `tailf:invocation-mode` substatement to `tailf:transaction-hook`, like this:

```
tailf:callpoint foocp {
  tailf:transaction-hook subtree {
    tailf:invocation-mode per-transaction;
  }
}
```

The `per-transaction` argument tells ConfD that this hook should only have one data callback invocation, regardless of the details of the changes to the objects the hook is assigned to. We can even use the same callpoint name, with the same `tailf:invocation-mode` statement, at several points in the data model, and still only get one callback invocation. The data callback that gets invoked for a transaction hook specified like this is called `write_all()` (see the confd_lib_dp(3) manual page). It is thus the only callback that should be registered for such a hook.

Since hook code gets to execute for all the possible write callbacks, the number of use cases for hook code is very large. One common use case is once again associated to the implementation of standard MIBS. Depending on the nature of the chosen standard MIBs, we may need to maintain mapping tables. If for example the keys differ in the SNMP table from the high level data model we may need to maintain additional mapping tables that are maintained by hook code.

10.6.1. Set Hooks and Candidate Configuration

When ConfD has been configured to provide a candidate configuration, set hook code will be invoked when changes are done to the candidate configuration, while transaction hooks will be invoked when the candidate is committed to running.

There are situations when you only want your hook to modify the running configuration:

- A hook can use maapi to modify config elements that the operator is not allowed to modify directly, according to the active aaa rule set. If such a modification is done on the candidate configuration store, the operator will not be allowed to commit the candidate configuration to the running configuration. In this situation you must thus use a transaction hook to modify the configuration.

10.7. Hidden Data

It is sometimes useful to hide nodes from some of the northbound interfaces. The `tailf:export` statement can be used to hide an entire namespace. More fine grained control can be attained with the `tailf:hidden` statement.

The `tailf:hidden` statement names a *hide group*, i.e. all containers and leafs that has the tailf:hidden statement, with a specific hide group, are treated the same way as far as being hidden or invisible. The hide group name `full` is given a special meaning. The `full` hide group is hidden from all northbound interfaces, not just `user` interfaces.

A node with the `tailf:hidden` statement must be optional or have a default value if it can be implicitly created via the creation of a differently hidden node higher up in the hierarchy (e.g. hidden leafs in a non-hidden list entry).

A related situation is when some nodes should be displayed to user only when a certain condition is met. For example, the `ethernet` subtree should be displayed only when the type of an interface is `ethernet`. This can be achieved through the `tailf:display-when` statement.

10.7.1. Fully Hidden Nodes

This is nodes that may be useful for the application code, but should be hidden from all northbound interfaces. An example is the set of physical network interfaces on a device and their types. This is static data, i.e. it can't be changed by configuration, but it can vary between different models of a device that run the same software, and the device-specific data can be provided via init file or through MAAPI.

This type of data could also be realized via a separate namespace where `tailf:export` is used to limit the visibility, but being able to have some nodes in the data model hidden while others are not allows for greater flexibility - e.g. list entries in the config data can have hidden containers or leafs, which get instantiated automatically along with the visible config nodes.

10.7.2. Hiding nodes from User Interfaces

This is data that is fully visible to programmatic northbound interfaces such as NETCONF, but normally hidden from user interfaces such as CLI and Web UI. Examples are data used for experimental or end-customer-specific features, similar to hidden commands in the CLI but for data nodes.

A user interface may give access to this type of data (and even totally hidden data) if the user executes an `unhide` command identifying the set of hidden data that should be revealed. After this these data nodes appear the same as unhidden data, i.e. they are included in tab completion, listed by **show** commands etc.

A hide group can only be *unhidden* if the group is listed in the confd.conf file. This means that a hide group will be completely hidden to the user interfaces unless it has been explicitly allowed to be unhidden in the confd.conf file. A password can optionally be required to unhide a group.

```
<hideGroup>
   <name>debug</name>
   <password>secret</password>
</hideGroup>
```

10.7.3. Conditional Display

A typical usage example is a discriminated union. One leaf is the type of something, and depending on the value of this leaf, different containers are visible:

```
container service {
    leaf type {
        default http;
        type enumeration {
            enum http;
            enum smtp;
        }
    }
    choice service-type {
        container http {
            presence "HTTP enabled";
            tailf:display-when '/service/type = "http"';
            leaf addr {
                mandatory  true;
                type inet:ipv4-address;
            }
            leaf docroot {
                mandatory  true;
```

```
                type string;
            }
        }
        container smtp {
            presence "SMTP enabled";
            tailf:display-when '/service/type = "smtp"';
            leaf smtp-relay {
                mandatory  true;
                type boolean;
            }
            leaf use-virtual-mbox {
                type boolean;
            }
        }
    }
}
```

In this example, the "smtp" container should be visible to the user only when the value of `service-type` is `smtp`.

This can be accomplished by using the `tailf:display-when` statement. It contains an XPath expression which specifies when the node should be displayed:

10.8. tailf:symlink

NOTE: The usage of symlink is not recommended. If it is used, use it carefully. It can be made to work in some special cases, but some other use cases, described below, are problematic. In the problematic cases, `tailf:link` can often be used instead. See Section 10.8.1, “Discussion” for more details.

Sometimes we want to move things in our data models. One major downside of moving structures around in the data model is that all code that reads data from CDB has to be changed. Sometimes this is not feasible because we do not want to change this code. In this case the `symlink` feature may be a solution.

Another situation where symlinks might be the remedy is when we want to move things in the data model due to aesthetics in the human interfaces, the CLI and the Web UI that are both rendered from the data model.

The syntax for a `symlink` is straightforward:

```
container top {
  tailf:symlink foo {
    tailf:path "/baz/bar";
  }
}

container baz {
  container bar {
    leaf target {
      type string;
      default "Zappa";
    }
  }
}
```

With the above construct we create a link from `/top/foo` to `/baz/bar`.

The net result of the link is that it will appear as if all data found in the data model below the link target, e.g. `/baz/bar` will appear under the link source, e.g. `/top/foo`. Thus if prior to the symlink the path `/baz/bar{13}/server{www}/port` was a valid key path, now the path `/top/foo/bar{13}/server{www}/port` point to the same configuration item. Hence we now have two different ways to configure the same item. The remedy to that is is typically to hide the the target configuration from the north bound agents using the `tailf:hidden` or `tailf:export` statements.

The textual format of a symlink is an XPath absolute location path.

The target of a symlink can contain instantiated keys. Using XPath notation we could point to the `counters` element in a specific `server` as in :

```
tailf:symlink mycounter {
  tailf:path "/servers/server[ip='10.0.0.1'][port=80]'
           + "/counter";
}
```

Slightly more advanced is that the target can refer to keys in the source. If the target lies within a list, all keys must be specified. A key either has a value, or is a reference to a key in the path of the source element, using the function `current()` as starting point for an XPath location path. Example of a valid target for a symlink is:

```
/servers/server[ip='10.0.0.1'][port=current()/../myport]/mycounter
```

This feature must be used when we want to use keys in our own path down the XML tree and use those very same keys to find an other way down the XML tree.

At link time, confdc checks that the target of the symlink can actually exist at run-time. A symlink may point to another symlink. confdc checks at link time, and ConfD at load time, that no cycles exist. Dangling pointers may occur though. If we have a symlink that points into an optional container or into a list entry, the target container can be removed. It is up to the application developers to check that dangling pointers do not occur. If they do, and the symlink is accessed, this is logged in the developer log as an error.

Finally we need to mention that when we are symlinking into another namespace, we must indicate that using the default XML notation for referencing elements in other namespaces. Thus if we want to have a symlink into the `http://example.com/ns/servers` namespace we typically have to import the module defining that namespace, and use the prefix of that module in the symlink as in:

```
import servers {
  prefix srv;
}
...

tailf:symlink myserver {
  tailf:path "/srv:servers/srv:server";
}
```

10.8.1. Discussion

Northbound Client Applications

The YANG modules that the device publishes to the northbound clients define the API between the client and the device. Specifically, the YANG module defines the complete data tree for configuration and operational state.

One problem with `tailf:symlink` is that a client application (EMS or NMS or similar) cannot just ignore the statement, since it defines a subtree in the datastore. But since this is not a standard statement, third-party clients will ignore it. This means that when they talk to the device, they will not understand the subtree under the symlink.

Note that symlink is used to provide a different entry point to the same underlying data. If both the data model with the symlink and the target data model is being exposed at the same time to the client, this will be confusing to the client. A change in one part of the tree makes another part change automatically.

One solution to this problem is to make sure that the module with the symlink is not exposed to the client. Instead, publish the target module only.

Another solution to this problem is to not publish the target module, but instead use **pyang** to sanitize the module with the symlink. The sanitizing process replaces the symlink statement with a copy of the target subtree. Then publish the sanitized version of the module.

NOTE: If a module is not published to a client, it should have a `tailf:export` statement, to exclude the protocol the client is using. For example, suppose we have a symlink in our module `acme-system.yang` from `/system/netconf/nacm` to the standard NACM path `/nacm`. The standard NACM module might be annotated with:

```
tailf:export netconf;
```

and the system module with:

```
tailf:export cli;
tailf:export webui;
```

Errors

Another problem with symlinks is if the target subtree contains `must` expression or validation code that refers to nodes outside the target subtree. For example, if a symlink `foo` points to `/x/z` in this example:

```
container x {
  leaf y {
    type int32;
  }
  leaf z {
    type int32;
    must ". > ../y" {
     error-message "x must be greater than y";
  }
}
```

If someone sets the `foo` to a value that happens to invalidate the must expression, the error message will be misleading since it refers to a node they cannot see, and it is not easily fixed, since they cannot modify the node y.

Chapter 11. Actions

11.1. Introduction

When we want to define operations that do not affect the configuration data store, we can use the `tailf:action` statement in the YANG data model. The action definition specifies how the action is invoked, including input and output parameters (if any). Once defined, the action is available for invocation from all of NETCONF, CLI and Web UI. The action can be implemented either as a callback function or as an executable. Action support is also discussed in Chapter 15, *The NETCONF Server*, Chapter 16, *The CLI agent*, and WebUI.

11.2. Action as a Callback

To specify that the action is implemented as a callback in an application Daemon, that registers with ConfD via the C API described in the confd_lib_dp(3) manual page, we use the `tailf:actionpoint` statement.

The application must register the callback by calling this function:

`int` **`confd_register_action_cbs`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_action_cbs *`*`acb`*`);`

The struct confd_action_cbs is defined as:

```
struct confd_action_cbs {
    char actionpoint[MAX_CALLPOINT_LEN];
    int (*init)(struct confd_user_info *uinfo);
    int (*abort)(struct confd_user_info *uinfo);
    int (*action)(struct confd_user_info *uinfo,
                  struct xml_tag *name,
                  confd_hkeypath_t *kp,
                  confd_tag_value_t *params,
                  int nparams);
    int (*command)(struct confd_user_info *uinfo,
                   char *path, int argc, char **argv);
    int (*completion)(struct confd_user_info *uinfo,
                      int cli_style, char *token, int completion_char,
                      confd_hkeypath_t *kp,
                      char *cmdpath, char *cmdparam_id,
                      struct confd_qname *simpleType, char *extra);
    void *cb_opaque;        /* private user data */
};
```

The `actionpoint` element gives the name of the actionpoint from the data model, and the `init` and `action` elements must point to two callback functions that are called in sequence when the action is invoked. In the `init()` callback, we must associate a worker socket with the action. This socket will be used for the invocation of the `action()` callback, which actually carries out the action. Thus in a multi threaded application, actions can be dispatched to different threads.

The `action()` callback is invoked with parameters pertaining to the action, in particular a hashed Keypath that can identify a particular `list` instance that the action should be applied to, and an array giving the input parameters. The parameters have the form of an XML instance document conforming to the specification in the `input` statement in the data model, and are represented as described for the Tagged Value Array format in the section called “XML STRUCTURES” in the confd_types(3) manual page. If the action should return any data values, it must call `confd_action_reply_values()` with an array of values in the same form, conforming to the specification in the `output` statement in the data model.

Unlike the callbacks for data and validation, there is no transaction associated with an action callback. However an action is always associated with a user session (NETCONF, CLI, etc), and only one action at a time can be invoked from a given user session.

See the section called “CONFD ACTIONS” in the confd_lib_dp(3) manual page for additional information about the action callbacks.

11.2.1. Example

As an example, we will look at one of several actions implemented in `intro/7-c_actions` in the ConfD examples collection. The data model defines a list of servers, and an action that allows us to request that a server is reset at some point in time. First, we specify the action in the data model like this:

```
    list server {
      key name;
      max-elements 64;
      leaf name {
        tailf:cli-allow-range;
        type string;
      }
      tailf:action reset {
        tailf:actionpoint reboot-point;
        input {
          leaf when {
            type string;
            mandatory true;
          }
        }
        output {
          leaf time {
            type string;
            mandatory true;
          }
        }
      }
    }
```

Our implementation must have an `init()` callback and an `action()` callback. The `init()` callback is straightforward:

```
static int init_action(struct confd_user_info *uinfo)
{
    int ret = CONFD_OK;

    printf("init_action called\n");
    confd_action_set_fd(uinfo, workersock);
    return ret;
}
```

I.e. it just tells ConfD that we want the previously connected worker socket to be used for the invocation of the `action()` callback. The calls to `printf(3)` in these callback functions are of course only for the purpose of illustration when we run the example. The `action()` callback, called `do_action()`, has a twist: we are using the same callback for multiple actions in the data model, and the code looks at the *`name`* parameter (the argument to `tailf:action` in the data model) to decide what to do:

```
/* This is the action callback function.  In this example, we have a
   single function for all four actions. */
```

```
static int do_action(struct confd_user_info *uinfo,
                     struct xml_tag *name,
                     confd_hkeypath_t *kp,
                     confd_tag_value_t *params,
                     int n)
{
    confd_tag_value_t reply[3];
    int i, k;
    char buf[BUFSIZ];
    char *p;

    printf("do_action called\n");

    for (i = 0; i < n; i++) {
        confd_pp_value(buf, sizeof(buf), CONFD_GET_TAG_VALUE(&params[i]));
        printf("param %2d: %9u:%-9u, %s\n", i, CONFD_GET_TAG_NS(&params[i]),
               CONFD_GET_TAG_TAG(&params[i]), buf);
    }

    switch (name->tag) {
```

Our "reset" action is handled in this branch:

```
    case config_reset:
        printf("reset\n");
        p = CONFD_GET_CBUFPTR(CONFD_GET_TAG_VALUE(&params[0]));
        i = CONFD_GET_BUFSIZE(CONFD_GET_TAG_VALUE(&params[0]));
        strncpy(buf, p, i);
        buf[i] = 0;
        strcat(buf, "-result");
        i = 0;
        CONFD_SET_TAG_STR(&reply[i], config_time, buf); i++;
        confd_action_reply_values(uinfo, reply, i);
        break;
```

The `when` leaf from the `input` statement in the data model is the first and only parameter, and is available to the callback in `params[0]`. A real implementation of "reset" would analyze the string, e.g. "now" for immediate reset or a date and time for some point in the future when the reset should be carried out. Here we just append "-result" to the string and return that as a reply for the `time` leaf in the `output` statement, by setting the first element in our `reply[]` array and calling `confd_action_reply_values()`. Finally we must register the callbacks:

```
    /* register the action handler callback */
    memset(&acb, 0, sizeof(acb));
    strcpy(acb.actionpoint, "reboot-point");
    acb.init = init_action;
    acb.action = do_action;
    acb.abort = abort_action;

    if (confd_register_action_cbs(dctx, &acb) != CONFD_OK)
        fail("Couldn't register action callbacks");
```

11.2.2. Using Threads

If we have long-running action callbacks (e.g. file download), it will typically be necessary to use multi-threading for the daemon that handles the callbacks. Without threads, only one invocation of a given action callback can be running at any point in time. Thus if the same action is requested from another user session, the request will block until the currently running invocation has completed.

Even if we do not want to allow multiple invocations of an action callback to run in parallel, having one thread for the control socket and one for the worker socket will make it possible to return an error from the action `init()` callback when we are "busy" with a running action, instead of having the user wait for the currently running action to complete. It will also allow us to handle other control socket requests promptly. In the general case, where we *do* want to handle multiple action callback requests in parallel, we need to use multiple worker sockets, with one thread handling each worker socket. We also need one thread to handle the control socket, dispatching the callback requests to the different worker sockets.

The strategy to use for creating and allocating the worker sockets and threads is up to the application, based on the needs for responsiveness in the user interface, resource usage requirements, and other application-specific considerations. We can set up a fixed pool of sockets and threads on startup, or we can connect worker sockets and spawn threads dynamically on demand, as well as close worker sockets and terminate threads that are no longer in use.

The `intro/9-c_threads` example in the ConfD examples collection demonstrates one such strategy, where worker sockets/threads for action callbacks are created on demand, giving each user session that requests an action its own dedicated action worker thread. The user session `stop()` callback (see `confd_register_usess_cb()` in the confd_lib_dp(3) manual page) is used to mark sockets/threads as "idle" and available for assignment as action workers for other user sessions. With this strategy we may need as many threads as there can be concurrent user sessions - by default there is no limit on this, but such limits can be configured in confd.conf (e.g. `/confdConfig/sessionLimits/maxSessions` for the total number of concurrent user sessions across all northbound interfaces), see the confd.conf(5) manual page.

11.3. Action as an Executable

To specify that the action is implemented as a standalone executable (this could be either a compiled program or a script), that is run to completion on each action invocation, we use the `tailf:exec` statement. This has several substatements specifying how the executable should be invoked - for the full details, see the tailf_yang_extensions(5) manual page:

`tailf:args`
: A space separated list of argument strings to pass to the executable when it is invoked by ConfD. It may contain variables on the form `$(variablename)`, that are expanded before the command is executed. E.g.

```
tailf:args "-p $(path)";
```

  will result in the first argument being "-p" and the second being the CLI form (space-separated elements) of the path leading to the action's parent container.

`tailf:uid`
: The user id to use when running the executable.

`tailf:gid`
: The group id to use when running the executable.

`tailf:wd`
: The working directory to use when running the executable. If not specified, the home directory of the user invoking the action is used, except for the case of a CLI session invoked via the **confd_cli** command - then the directory where **confd_cli** was invoked is used.

`tailf:global-no-duplicate`
: Specifies that only one instance with the name that is given as argument can be run at any time in the system.

`tailf:interrupt`
Specifies which signal to use to interrupt the executable when the action invocation is interrupted.

The input parameters are passed to the executable as arguments (following those specified by `tailf:args`) in the same general form as for a callback invocation. Each tag-value pair results in two arguments, the first is the tag name as a string, the second is the value in string form. The special elements used to indicate the start and end of a list entry or container and a typeless leaf (i.e. C_XMLBEGIN, C_XMLEND, and C_XMLTAG in the C API) have the "values" __BEGIN, __END, and __LEAF. If the action has output parameters, their values should be printed on standard output in the same form.

If the execution is successful, the executable should exit with code 0. Otherwise it may print error information on standard output before exiting with a non-zero code. The error information can be either a free-form string (corresponding to the `confd_action_seterr()` function for the callback) or structured information corresponding to the NETCONF form described in the section called “EXTENDED ERROR REPORTING” in the confd_lib_lib(3) manual page. It could for example print this on standard output:

```
error-tag resource-denied
error-message "out of memory"
```

The `error-tag` element is required in this case.

11.3.1. Example

Another action in `intro/7-c_actions` in the examples collection is implemented as a Perl script. Here the data model defines a list of hosts, and an action that allows us to send **ping** requests to a host. We specify the action in the data model like this:

```
    list host {
      key name;
      leaf name {
        type string;
      }
      tailf:action ping {
        tailf:exec "./ping.pl" {
          tailf:args "-c $(context) -p $(path)";
        }
        input {
          leaf count {
            type int32;
            default "3";
          }
        }
        output {
          leaf header {
            type string;
          }
          list response {
            leaf data {
              type string;
            }
          }
          container statistics {
            leaf packet {
              type string;
            }
            leaf time {
              type string;
            }
```

```
          }
        }
      }
    }
```

The "-c $(context) -p $(path)" argument for the `tailf:args` statement has the effect that the context (cli, netconf, etc) and the data model path will be passed to the script, followed by the `count` parameter from the `input` statement in the data model. This parameter may have been given by the user or defaulted according to the data model. Thus, if a host called "earth" exists in the configuration, and we use the J-CLI and type this in operational mode:

```
request config host earth ping count 5
```

Then the script will be invoked as:

```
./ping.pl -c cli -p 'config host earth' count 5
```

The script starts by parsing these arguments, in particular picking up the last word of the `-p` value for the host name (stored in the `$host` variable) and the argument for the *`count`* parameter (stored in the `$count` variable):

```
while ($#ARGV >= 0) {
    if ($ARGV[0] eq "-c") {
        $context = $ARGV[1];
        shift; shift;
    } elsif ($ARGV[0] eq "-p") {
        @path = split(' ', $ARGV[1]);
        shift; shift;
    } elsif ($ARGV[0] eq "count") {
        $count = $ARGV[1];
        shift; shift;
    } else {
        &fail("Unknown argument " . $ARGV[0]);
    }
}
$host = $path[$#path];
```

In this example, the input parameters are very simple, just a single leaf. If we have lists or containers for `input` in the data model, the script will receive them with __BEGIN and __END "values", as shown for the output below.

Having collected the required parameters, and taking some OS dependencies into account, the script proceeds to run the actual **ping** command, collecting the output (standard output and standard error) in the `$out` variable, and checking the exit code. On a non-zero exit code (indicating failure), the `fail` function will just print the output from **ping** on standard output and exit with code 1.

```
$ENV{'PATH'} = "/bin:/usr/bin:/sbin:/usr/sbin:" . $ENV{'PATH'};
if (`uname -s` eq "SunOS\n") {
    $cmd = "ping -s $host 56 $count";
} else {
    $cmd = "ping -c $count $host";
}
$out = `$cmd 2>&1`;
if ($? != 0) {
    &fail($out);
}
```

If the execution of **ping** is successful, the script splits the output into lines and generates a reply according to the `output` statement in the data model: each leaf is output with the leaf name followed by the value of

the leaf, and __BEGIN and __END "values" indicate the the start and end of each entry in the `response` list, as well as the start and end of the `statistics` container:

```
@result = split('\n', $out);
print "header 'Invoked from " . $context . ": " . $result[0] . "'\n";
for ($i = 0; $i < $count; $i++) {
    print "response __BEGIN data '" . $result[$i+1] . "' response __END\n";
}
$packets = $result[$#result-1];
$times = $result[$#result];
print "statistics __BEGIN\n";
print "packet '" . $packets . "' time '" . $times . "'\n";
print "statistics __END\n";

exit 0;
```

If we run the script interactively, using the command line above, we will get output that looks something like this:

```
$ ./ping.pl -c cli -p 'config host earth' count 5
header 'Invoked from cli: PING earth.tail-f.com (192.168.1.42): 56 data bytes'
response __BEGIN data '64 bytes from 192.168.1.42: icmp_seq=0 ttl=64 time=0.187 ms' respons
response __BEGIN data '64 bytes from 192.168.1.42: icmp_seq=1 ttl=64 time=0.150 ms' respons
response __BEGIN data '64 bytes from 192.168.1.42: icmp_seq=2 ttl=64 time=0.208 ms' respons
response __BEGIN data '64 bytes from 192.168.1.42: icmp_seq=3 ttl=64 time=0.205 ms' respons
response __BEGIN data '64 bytes from 192.168.1.42: icmp_seq=4 ttl=64 time=0.204 ms' respons
statistics __BEGIN
packet '5 packets transmitted, 5 packets received, 0.0% packet loss' time 'round-trip min/a
statistics __END
```

ConfD will parse this output and deliver the data to the requesting northbound agent. Since the values include whitespace, they must be enclosed in quotes - either single quotes (') as in this example or double quotes (") can be used. Arbitrary whitespace can be used to separate node names and values.

11.4. Related functionality

The action invocation mechanism is also used for some other related purposes:

- A NETCONF RPC (see Section 15.5, “Extending the NETCONF Server” in Chapter 15, *The NETCONF Server*) can be specified to invoke a callback or an executable (where ConfD translates the XML), via the `tailf:actionpoint` and `tailf:exec` statements, respectively. This is implemented via invocation of the `action()` callback, or running of an executable, exactly as described above. (When the `tailf:raw-xml` statement is used with `tailf:exec`, the argument and result passing described above is not applicable.)

- The CLI can invoke "capi callbacks" for either complete CLI commands or command completion functionality (see Chapter 16, *The CLI agent*). This is implemented via invocation of `command()` and `completion()` callbacks, respectively, that are registered by the application in the same way as an `action()` callback. See the section called “CONFD ACTIONS” in the confd_lib_dp(3) manual page for the details.

Chapter 12. Notifications

12.1. ConfD Asynchronous Events

ConfD can deliver various classes of events to subscribing applications. The architecture is based on notification sockets. The application(s) connect a notifications socket to ConfD. The application provides a bit mask indicating which types of events the application is interested in. The application polls the socket and invokes the API function `confd_read_notification()` whenever the socket is ready to read. The API function populates a `struct confd_notification` structure.

The following is a list of the different asynchronous event classes that can be delivered from ConfD to the application(s). See also the confd_lib_events(3) manual page. The program `misc/notifications/confd_notifications.c` in the examples collection illustrates subscription and processing for all these events, and can also be used standalone in a development environment to monitor ConfD events.

- `CONFD_NOTIF_AUDIT` - Audit events.
- `CONFD_NOTIF_AUDIT_SYNC` - Indicates that audit notifications (CONFD_NOTIF_AUDIT) must be synced by the application.
- `CONFD_NOTIF_DAEMON` - Syslog events that also go to `/confdConf/logs/confdLog`.
- `CONFD_NOTIF_NETCONF` - Syslog events that also go to `/confdConf/logs/netconfLog`.
- `CONFD_NOTIF_DEVEL` - Syslog events that also go to `/confdConf/logs/developerLog`.
- `CONFD_NOTIF_TAKEOVER_SYSLOG` - Syslog control.
- `CONFD_NOTIF_COMMIT_SIMPLE` - Commit message.
- `CONFD_NOTIF_COMMIT_DIFF` - A complete diff compared to previous configuration.
- `CONFD_NOTIF_COMMIT_FAILED` - Possible data inconsistency event.
- `CONFD_NOTIF_CONFIRMED_COMMIT` - Events concerning confirmed commit processing.
- `CONFD_NOTIF_COMMIT_PROGRESS` - Events with commit progress information.
- `CONFD_NOTIF_USER_SESSION` - Whenever a user session is started or stopped.
- `CONFD_NOTIF_HA_INFO` - Changes in ConfD's perception of the cluster configuration.
- `CONFD_NOTIF_HA_INFO_SYNC` - Indicates that HA notifications (CONFD_NOTIF_HA_INFO) must be synced by the application.
- `CONFD_NOTIF_SUBAGENT_INFO` - Subagent related events.
- `CONFD_NOTIF_SNMPA` - SNMP agent audit log.
- `CONFD_NOTIF_FORWARD_INFO` - Events related to forwarding (proxying) of northbound agents.
- `CONFD_NOTIF_UPGRADE_EVENT` - Events generated for in-service upgrade.
- `CONFD_NOTIF_HEARTBEAT` - Heartbeat events.

- `CONFD_NOTIF_HEALTH_CHECK` - Health check events.
- `CONFD_NOTIF_STREAM_EVENT` - Notification stream events.

12.2. Audit Messages

Many applications need explicit control over where and in which format the various audit messages are sent. By audit messages here we mean any message related to user login/logout/reconfig activity. The list of different audit messages that are possible to receive can be found in the file `confd_logsyms.h`

In order to receive the audit message we must first connect a notifications socket.

Example 12.1. Creating a notification socket

```
confd_init("Foobar", stderr, debuglevel);
if ((notsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
    confd_fatal("Failed to open notsocket\n");

inet_aton("127.0.0.1", &in);
addr.sin_addr.s_addr = in.s_addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(CONFD_PORT);
dflag = CONFD_NOTIF_AUDIT;
if (confd_notifications_connect(
        notsock,
        (struct sockaddr*)&addr,
        sizeof (struct sockaddr_in), dflag) < 0 ) {
    confd_fatal("Failed to confd_connect() to confd \n");
}
```

The `dflags` argument is bit mask indicating which classes of notifications messages we wish to receive over the socket. It is possible to receive several different classes of notifications messages over the same socket.

Once we have the socket setup, we add it to our pollset and invoke `confd_read_notification()` once the socket is ready to read.

Example 12.2. reading the audit data

```
while (1) {
     struct pollfd set[1];
     struct confd_notification n;
     set[0].fd = notsock;
     set[0].events = POLLIN;
     set[0].revents = 0;

     if (poll(&set[0], 1, -1) < 0) {
         perror("Poll failed:");
         continue;
     }

     if (set[0].revents & POLLIN) {
         if (confd_read_notification(notsock, &n) != CONFD_OK)
             exit(1);
         switch(n.type) {
         case CONFD_NOTIF_AUDIT:
             printf("audit: sym=%d, user=%s/%d %s\n",
```

```
                    n.n.audit.logno,
                    n.n.audit.user, n.n.audit.usid, n.n.audit.msg);
            break;
            .......
```

The structure struct confd_notification is defined as:

```
enum confd_notification_type {
    CONFD_NOTIF_AUDIT                   = (1 << 0),
    CONFD_NOTIF_DAEMON                  = (1 << 1),
    CONFD_NOTIF_TAKEOVER_SYSLOG         = (1 << 2),
    CONFD_NOTIF_COMMIT_SIMPLE           = (1 << 3),
    CONFD_NOTIF_COMMIT_DIFF             = (1 << 4),
    CONFD_NOTIF_USER_SESSION            = (1 << 5),
    CONFD_NOTIF_HA_INFO                 = (1 << 6),
    CONFD_NOTIF_SUBAGENT_INFO           = (1 << 7),
    CONFD_NOTIF_COMMIT_FAILED           = (1 << 8),
    CONFD_NOTIF_SNMPA                   = (1 << 9),
    CONFD_NOTIF_FORWARD_INFO            = (1 << 10),
    CONFD_NOTIF_NETCONF                 = (1 << 11),
    CONFD_NOTIF_DEVEL                   = (1 << 12),
    CONFD_NOTIF_HEARTBEAT               = (1 << 13),
    CONFD_NOTIF_CONFIRMED_COMMIT        = (1 << 14),
    CONFD_NOTIF_UPGRADE_EVENT           = (1 << 15),
    CONFD_NOTIF_COMMIT_PROGRESS         = (1 << 16),
    CONFD_NOTIF_AUDIT_SYNC              = (1 << 17),
    CONFD_NOTIF_HEALTH_CHECK            = (1 << 18),
    CONFD_NOTIF_STREAM_EVENT            = (1 << 19),
    CONFD_NOTIF_HA_INFO_SYNC            = (1 << 20),
    NCS_NOTIF_PACKAGE_RELOAD            = (1 << 21),
    NCS_NOTIF_CQ_PROGRESS               = (1 << 22)
};

struct confd_notification {
    enum confd_notification_type type;
    union {
        struct confd_audit_notification audit;
        struct confd_syslog_notification syslog;
        struct confd_commit_notification commit;
        struct confd_commit_diff_notification commit_diff;
        struct confd_user_sess_notification user_sess;
        struct confd_ha_notification hnot;
        struct confd_subagent_notification subagent;
        struct confd_forward_notification forward;
        struct confd_commit_failed_notification cfail;
        struct confd_snmpa_notification snmpa;
        struct confd_confirmed_commit_notification confirm;
        struct confd_upgrade_notification upgrade;
        struct confd_progress_notification progress;
        struct confd_stream_notification stream;
        struct ncs_cq_progress_notification cq_progress;
    } n;
};
```

Where the field `type` indicates the type of the message. Depending on the type, one of the other union structures is populated by the `confd_read_notification()` API function

In our case with audit messages, we get a struct confd_audit_notification structure populated.

```
struct confd_audit_notification {
```

```
    int logno;   /* number from confd_logsyms.h */
    char user[MAXUSERNAMELEN];
    char msg[BUFSIZ];
    int usid;   /* session id (0 means - not applicable ) */
};
```

The `logno` is an integer which defines the event. All log and audit events generated by confd are enumerated and documented in the include file `confd_logsyms.h`.

If we have indicated that we want to synchronize audit messages with ConfD, we must call `confd_sync_audit_notification()` after receiving an audit message, to signal ConfD that it can continue processing.

12.3. Syslog Messages

Some applications have explicit requirements not only where to send syslog messages (this can be easily configured in `confd.conf`) but also how and on which format to send the syslog messages. By default, ConfD will simply invoke the standard libc `syslog()` function.

It is possible to subscribe to ConfD syslog messages and also at the same time suppress ConfD's own syslogging. To subscribe to syslog messages, the application needs to use one or more of the flags `CONFD_NOTIF_DAEMON`, `CONFD_NOTIF_NETCONF`, and `CONFD_NOTIF_DEVEL` in the mask given to `confd_notifications_connect()`.

If the mask given to `confd_notifications_connect()` contains the flag `CONFD_NOTIF_TAKEOVER_SYSLOG`, ConfD will not invoke the regular `syslog()` function. Thus in this case, it is entirely up to the application to actually report the messages.

If all notifications subscribers that have requested the `CONFD_NOTIF_TAKEOVER_SYSLOG` feature close their notifications sockets, ConfD will revert to the behavior of invoking libc `syslog()`. Similarly, when ConfD is starting, before any application processes has connected and requested the `CONFD_NOTIF_TAKEOVER_SYSLOG` feature, ConfD will of course use the standard `syslog()` functionality

When subscribing to syslog messages we receive a populated struct confd_syslog_notification structure:

```
struct confd_syslog_notification {
    int prio;   /* from syslog.h */
    int logno;  /* number from confd_logsyms.h */
    char msg[BUFSIZ];
};
```

The `logno` is an integer which defines the event. All syslog and audit events generated by confd are enumerated and documented in the include file `confd_logsyms.h`.

12.4. Commit Events

There are two different types of commit events we can subscribe to. One really simple which just indicates that a commit from a north bound agent has occurred. This is achieved by setting the subscription bitmask to contain the flag: `CONFD_NOTIF_COMMIT_SIMPLE`. The message we receive contains a struct confd_commit_notification structure:

```
struct confd_commit_notification {
    enum confd_dbname database;
    int diff_available;
    struct confd_user_info uinfo;
```

```
    int flags;
};
```

This just provides information on which user committed to which database e.g. running or the candidate. The other commit notification is considerably more complex and it provides information on exactly which nodes were changed.

The flag value is `CONFD_NOTIF_COMMIT_DIFF`, and the structure we receive is:

```
struct confd_commit_diff_notification {
    enum confd_dbname database;
    struct confd_user_info uinfo;
    struct confd_trans_ctx  *tctx;
    int flags;
};
```

The structure contains a transaction context which we can choose to use with `maapi_attach()` and thus attach to the currently executing transaction. When the event is generated, this transaction has successfully been committed by all data providers, but the commit operation has not completed and it is hanging, waiting for the application to invoke `confd_diff_notification_done()`.

`maapi_attach()` attaches a transaction context. We can then use that transaction context to read from the transaction. The transaction has a list of nodes which constitute the configuration changes in the transaction. We can traverse this list using the function `maapi_diff_iterate()` which will invoke a user supplied function for each and every modification in the transaction.

The purpose of this feature is not to be able to check the commit diff. All such checking should be done using the normal validation routines. The purpose is rather to be able to log diffs on a per commit basis.

Thus the first thing we need if we want to traverse the diff list is a function to be invoked for every diff item. Our example here will just format the data and print to stdout.

```
static enum maapi_iter_ret iter(confd_hkeypath_t *kp,
                                enum maapi_iter_op op,
                                confd_value_t *oldv,
                                confd_value_t *v,
                                void *state)
{
    char path[BUFSIZ];
    char value[BUFSIZ];
    char *opstr;
    struct confd_cs_node *node;
    confd_hkeypath_t *dkp;
    int i;

    confd_pp_kpath(path, sizeof(path), kp);
    value[0] = 0;
    switch (op) {
    case MOP_CREATED:
        opstr = "created";
        break;
    case MOP_DELETED:
        opstr = "deleted";
        break;
    case MOP_MODIFIED:
        opstr = "modified";
        break;
    case MOP_VALUE_SET:
        opstr = "value_set";
```

```
        node = confd_find_cs_node(kp, kp->len);
        confd_val2str(node->info.type, v, value, sizeof(value));
        break;
    case MOP_MOVED_AFTER:
        if (v == NULL) {
            opstr = "moved first";
        } else {
            opstr = "moved after";
            /* create+print a hkeypath for the entry this one was moved after */
            dkp = confd_hkeypath_dup(kp);
            for (i = 0; v[i].type != C_NOEXISTS; i++) {
                confd_free_value(&dkp->v[0][i]);
                confd_value_dup_to(&v[i], &dkp->v[0][i]);
            }
            confd_pp_kpath(value, sizeof(value), dkp);
            confd_free_hkeypath(dkp);
        }
        break;
    case MOP_ATTR_SET:
        if (v[1].type == C_NOEXISTS) {
            opstr = "attr_del";
            snprintf(value, sizeof(value), "%s", attr_str(&v[0]));
        } else {
            opstr = "attr_set";
            i = snprintf(value, sizeof(value), "%s -> ", attr_str(&v[0]));
            confd_pp_value(&value[i], sizeof(value) - i, &v[1]);
        }
        break;
    }
    printf ("ITER %s %s %s\n", path, opstr, value);
    return ITER_RECURSE;
}
```

The iteration function must return an enum maapi_iter_ret indicating to ConfD what to continue to do. We have the following possible return values:

- `ITER_STOP` - Stop. Do not invoke the iteration function any more for this transaction
- `ITER_RECURSE` - Iteration continues with all children of the modified node.
- `ITER_CONTINUE` - Iteration ignores the children of the node and continues with the node's sibling.

The iteration function is called for each modified node in the configuration. See the description of `maapi_diff_iterate()` in confd_lib_maapi(3) for a detailed description of when the different *op* values `MOP_CREATED`, `MOP_DELETED`, `MOP_MODIFIED`, `MOP_VALUE_SET`, and `MOP_MOVED_AFTER` are used.

Finally we must have a function which is invoked whenever we receive a notification of type `CONFD_NOTIF_COMMIT_DIFF`. The function must use the supplied transaction context and attach, and when it is done traversing the diff it must call `confd_diff_notification_done()`.

```
static void handle_diff_notif(struct confd_trans_ctx *tctx)
{
    /* first we need a maapi socket */
    int maapi_socket;

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

```
    if (maapi_connect(maapi_socket, (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

    /* no namespace needed for this */
    OK(maapi_attach(maapi_socket, -1, tctx));

    /* Now we can iterate through the currently hanging transaction */
    /* and read out all the diffs */
    OK(maapi_diff_iterate(maapi_socket, tctx->thandle, iter,
                          ITER_WANT_ATTR, NULL));

    /* and finally call done to release data and let  */
    /* the transaction finish */

    OK(confd_diff_notification_done(notif_socket, tctx));
    close(maapi_socket);
}
```

12.5. Commit Failure Events

The `CONFD_NOTIF_COMMIT_FAILED` event is generated when a data provider fails in its commit callback. ConfD executes a two-phase commit procedure towards all data providers when committing transactions. When a provider fails in commit, the system is an unknown state. See confd_lib_maapi(3) and the function `maapi_get_running_db_status()`. If the provider is "external", the name of the failing daemon is provided. If the provider is another NETCONF agent, the IP address and port of that agent is provided.

12.6. Confirmed Commit Events

When a a user has started a confirmed commit, when a confirming commit is issued, or when a confirmed commit is aborted, a `CONFD_NOTIF_CONFIRMED_COMMIT` event is generated. The application receives a struct confd_confirmed_commit_notification, which gives the specific action and user session info for the committer. For a confirmed commit, the timeout value is also given.

12.7. Commit Progress Events

By subscribing to the `CONFD_NOTIF_COMMIT_PROGRESS` event, the application can receive the same commit progress information that is reported when the **commit | details** CLI command is used. The application receives a struct confd_progress_notification structure.

12.8. User Sessions

We can get notifications on user sessions and on user session events. A user session corresponds to an actual user logging in to the system, for example a NETCONF manager

The struct confd_user_sess_notification structure is defined as:

```
enum confd_user_sess_type {
    CONFD_USER_SESS_START = 1,       /* a user session is started */
    CONFD_USER_SESS_STOP = 2,
    CONFD_USER_SESS_LOCK = 3,        /* a database is locked */
    CONFD_USER_SESS_UNLOCK = 4,
    CONFD_USER_SESS_START_TRANS = 5, /* a database transaction is started */
```

```
    CONFD_USER_SESS_STOP_TRANS = 6
};

struct confd_user_sess_notification {
    enum confd_user_sess_type type;
    struct confd_user_info uinfo;
    enum confd_dbname database;
};
```

This means that we can follow the progress of a user session, which databases are touched by the session etc.

12.9. High Availability - Cluster Events

ConfD HA capabilities are described in Chapter 23, *High Availability*. This section describes the various events that are asynchronously produced by ConfD when the cluster configuration is changed. These changes may be induced explicitly by the application through invocation of the various HA related API functions in `libconfd` or they may be induced by ConfD itself when the sockets between the HA nodes get closed. It is vital that the High-Availability-Framework (HAFW) subscribes to these messages and acts accordingly.

The struct confd_notification structure received by `confd_read_notification()` will populate the `hnot` field with a struct confd_ha_notification. This in its turn is yet another union structure with a type field.

```
struct confd_ha_notification {
    enum confd_ha_info_type type;
    /* additional info for various info types */
    union {
        int nomaster;                       /* CONFD_HA_INFO_NOMASTER  */
        struct confd_ha_node slave_died;    /* CONFD_HA_INFO_SLAVE_DIED */
        struct confd_ha_node slave_arrived;/* CONFD_HA_INFO_SLAVE_ARRIVED*/
        int cdb_initialized_by_copy;        /* CONFD_HA_INFO_SLAVE_INITIALIZED */
        int beslave_result;                 /* CONFD_HA_INFO_BESLAVE_RESULT */
    } data;
};
```

We start with a listing of types of the different HA related events that ConfD can send to the subscribing application. The enum is defined as:

```
enum confd_ha_info_type {
    CONFD_HA_INFO_NOMASTER          = 1,   /* we have no master */
    CONFD_HA_INFO_SLAVE_DIED        = 2,   /* a slave disappeared */
    CONFD_HA_INFO_SLAVE_ARRIVED     = 3,   /* a slave arrived to us */
    CONFD_HA_INFO_SLAVE_INITIALIZED = 4,   /* CDB is initialized */
    CONFD_HA_INFO_IS_MASTER         = 5,   /* we are now master */
    CONFD_HA_INFO_IS_NONE           = 6,   /* we are now none */
    CONFD_HA_INFO_BESLAVE_RESULT    = 7    /* result of async beslave() */
};
```

Each of the different informational messages has additional data associated to it.

- `CONFD_HA_INFO_NOMASTER` A node (which is a slave node) has lost contact with the master and is now in HA state `CONFD_HA_STATE_NONE`. Only sent on the slave node.

  Whenever we receive this message the `nomaster` field is populated. This is either the integer `CONFD_ERR_HA_CLOSED` if the slave lost contact with master due to the socket getting closed or the integer `CONFD_ERR_HA_NOTICK` if the slave has not received any live ticks from the master.

- CONFD_HA_INFO_SLAVE_DIED A master node lost contact with a slave node. Only sent on the master node. The field slave_died is populated with a struct confd_ha_node indicating which particular slave died.
- CONFD_HA_INFO_SLAVE_ARRIVED A master node was connected to by a slave node. Authentication was ok and the slave is initializing its CDB database. Only sent at the master node. The field slave_arrived is populated with a struct confd_ha_node indicating which slave arrived.
- CONFD_HA_INFO_SLAVE_INITIALIZED A slave node has just finished its initialization and synchronization of the database. The slave is now fully operational. Only sent at slave nodes. The field cdb_initialized_by_copy is set to *1* if ConfD concluded that the entire CDB database has to be copied and *0* if a copy was avoided.
- CONFD_HA_INFO_IS_MASTER The node has been successfully elevated to master. This is only sent at the master node, i.e. the node that just became master.
- CONFD_HA_INFO_IS_NONE The node has been set to *NONE* mode.
- CONFD_HA_INFO_BESLAVE_RESULT If we use asynchronous invocation of the confd_ha_beslave() function, i.e. with the parameter *waitreply* set to 0, this message is sent when the operation has completed. The field beslave_result is set to indicate the result which would have been returned by a synchronous invocation of confd_ha_beslave(). Thus if beslave_result is 0, the node has successfully become a slave, otherwise beslave_result is one of the confd_errno values that can be returned by synchronous invocation of confd_ha_beslave().

If we have indicated that we want to synchronize HA messages with ConfD, we must call confd_sync_ha_notification() after receiving a HA message, to signal ConfD that it can continue processing.

12.10. Subagent Events

The subagent mechanism is described in Chapter 25, *Subagents and Proxies*. This section describes the related events which ConfD generates when acting as a master agent.

When the notification type is CONFD_NOTIF_SUBAGENT_INFO, the struct confd_notification structure received by confd_read_notification() will populate the subagent field with a struct confd_subagent_notification.

```
struct confd_subagent_notification {
    enum confd_subagent_info_type type;
    char name[MAXAGENTNAMELEN];
};
```

The type field is one of the values CONFD_SUBAGENT_INFO_UP or CONFD_SUBAGENT_INFO_DOWN.

At first, each subagent is marked as being down. When ConfD successfully communicates with a subagent, it is marked as up, and a corresponding event is generated. A down event is generated only if ConfD tries to communicate with a subagent, but fails. Thus, if a subagent closes an idle connection to the master agent, it is not marked as down.

12.11. SNMP Agent Audit Log

The SNMP agent log is activated through the /confdCfg/logs/snmpLog element in the confd.conf configuration file.

The SNMP audit log messages can also be received and processed by an external C program over a notification socket. The application receives a struct confd_snmpa_notification structure. The structure contains a series of fields describing the sent or received SNMP PDU. It also contains a list of all varbinds in the PDU.

Each varbind contains a confd_value_t with the string representation of the SNMP value. Thus the type of the value in a varbind is always `C_BUF`. See `confd_events.h` include file for the details of the received structure.

The following code exemplifies how we write a program which establishes a notification socket and subscribes to all SNMP PDUs in and out of the system.

We start off with some auxiliary function to format the PDU type and the type of a "varbind"

```
char *vb_type(struct confd_snmp_varbind *vb) {
    switch (vb->vartype) {
    case CONFD_SNMP_NULL: return "NULL";
    case CONFD_SNMP_INTEGER: return "INTEGER";
    case CONFD_SNMP_Interger32: return "Integer32";
    case CONFD_SNMP_OCTET_STRING: return "OCTET STRING";
    case CONFD_SNMP_OBJECT_IDENTIFIER: return "OBJECT IDENTIFIER";
    case CONFD_SNMP_IpAddress: return "IpAddress";
    case CONFD_SNMP_Counter32: return "Counter32";
    case CONFD_SNMP_TimeTicks: return "TimeTicks";
    case CONFD_SNMP_Opaque: return "Opaque";
    case CONFD_SNMP_Counter64: return "Counter64";
    case CONFD_SNMP_Unsigned32: return "Unsigned32";
    }
    return "";
}

char *pdutype(struct confd_snmpa_notification *snmp) {
    switch (snmp->pdu_type) {
    case CONFD_SNMPA_PDU_V1TRAP: return("V1TRAP");
    case CONFD_SNMPA_PDU_V2TRAP: return("V2TRAP");
    case CONFD_SNMPA_PDU_INFORM: return("INFORM");
    case CONFD_SNMPA_PDU_GET_RESPONSE: return("GET_RESPONSE");
    case CONFD_SNMPA_PDU_GET_REQUEST: return("GET_REQUEST");
    case CONFD_SNMPA_PDU_GET_NEXT_REQUEST: return("GET_NEXT_REQUEST");
    case CONFD_SNMPA_PDU_REPORT: return("REPORT");
    case CONFD_SNMPA_PDU_GET_BULK_REQUEST: return("GET_BULK_REQUEST");
    case CONFD_SNMPA_PDU_SET_REQUEST: return("SET_REQUEST");
    default: return "";
    }
}
```

Following that we show the code which invokes `confd_read_notification()` and reads a C structure of the type struct confd_snmpa_notification

The structure contains the type of the PDU, various other fields and also the complete SNMP "varbind" lists in the PDU. The code prints the PDU type and then loops through all the varbinds and prints the value of each varbind.

```
    if (confd_read_notification(notsock, &n) != CONFD_OK)
        exit(1);
    switch(n.type) {
```

```
case CONFD_NOTIF_SNMPA: {
    int i,j;

    char buf[BUFSIZ];
    buf[0] = 0;
    char *ptr = &buf[0];
    struct confd_snmpa_notification *snmp = &n.n.snmpa;
    ptr += sprintf(ptr, "%s ", pdutype(snmp));
    ptr += sprintf(ptr,"Id = %d ", snmp->request_id);
    struct confd_ip *ip = &(snmp->ip);
    ptr += sprintf(ptr, " %s:%d ",
                   inet_ntoa(ip->ip.v4),
                   snmp->port);
    if ((snmp->error_status !=0 || snmp->error_index != 0)) {
        ptr += sprintf(ptr, "ErrIx = %d ", snmp->error_index);
    }
    else if (snmp->pdu_type == CONFD_SNMPA_PDU_V1TRAP) {
        ptr += sprintf(ptr,"Generic=%d Specific=%d",
                       snmp->v1_trap->generic_trap,
                       snmp->v1_trap->specific_trap);
        struct confd_snmp_oid *enterp = &snmp->v1_trap->enterprise;
        ptr += sprintf(ptr, " Enterprise=");
        for(i=0; i < enterp->len; i++) {
            ptr += sprintf(ptr,".%d", enterp->oid[i]);
        }
    }
    for (i=0; i < snmp->num_variables; i++) {
        struct confd_snmp_varbind *vb = &snmp->vb[i];
        ptr += sprintf(ptr,"\n   ");
        switch (vb->type) {
        case CONFD_SNMP_VARIABLE:
            ptr += sprintf(ptr, " %s ", vb_type(vb));
            ptr += sprintf(ptr,"%s=", vb->var.name);
            break;
        case CONFD_SNMP_OID:
            ptr += sprintf(ptr, " %s ", vb_type(vb));
            for (j=0; j < vb->var.oid.len; j++) {
                ptr += sprintf(ptr,"%d", vb->var.oid.oid[j]);
                if (j != vb->var.oid.len-1)
                    ptr += sprintf(ptr,".");
            }
            break;
        case CONFD_SNMP_COL_ROW:
            ptr += sprintf(ptr, " %s ", vb_type(vb));
            ptr += sprintf(ptr, "%s", vb->var.cr.column);
            for(j=0; j<vb->var.cr.rowindex.len; j++) {
                ptr += sprintf(ptr,".%d",
                               vb->var.cr.rowindex.oid[j]);
            }
            break;
        }
        if (vb->val.type == C_BUF) {
            char buf2[BUFSIZ];
            confd_pp_value(buf2, BUFSIZ, &vb->val);
            ptr += sprintf(ptr, "=%s", buf2);
        }
    }
    printf("%s\n\n", buf);
    confd_free_notification(&n);
}
```

12.12. Forwarding Events

ConfD can forward (proxy) connections from northbound agents. When forwarding starts, ends, or fails, a `CONFD_NOTIF_FORWARD_INFO` event is generated. The application receives a struct confd_forward_notification structure which gives the type of forwarding event, the name of the target for the forwarding, and user session information for the user that requested the forwarding.

12.13. In-service Upgrade Events

During in-service upgrade, the CONFD_NOTIF_UPGRADE_EVENT event is generated with different values for the enum confd_upgrade_event_type event. The events correspond to the different phases of the upgrade, see Chapter 13, *In-service Data Model Upgrade* and confd_lib_maapi(3) for a detailed description.

12.14. Heartbeat and Health Check Events

The `CONFD_NOTIF_HEARTBEAT` and `CONFD_NOTIF_HEALTH_CHECK` events can be used by applications that wish to monitor the health and liveness of ConfD itself. See confd_lib_events(3) for more details about this.

12.15. Notification stream Events

The `CONFD_NOTIF_STREAM_EVENT` event is generated for a notification stream, i.e. event notifications sent by an application as described in the section called “NOTIFICATION STREAMS” of confd_lib_dp(3). See confd_lib_events(3) for more details about this.

Chapter 13. In-service Data Model Upgrade

13.1. Introduction

When we want to change the data model used by ConfD, the simplest method is to stop and restart ConfD with the new .fxs files in place. CDB will then detect the change and perform an upgrade, automatically or assisted by external programs, as described in Chapter 5, *CDB - The ConfD XML Database*.

If it is necessary that ConfD keeps running throughout the data model change, we can instead control the upgrade from an external program using a set of MAAPI functions, as described in this chapter. The CDB upgrade will be performed in this case too of course, and all the techniques described in the CDB chapter are applicable here too. But in addition, this procedure requires careful synchronization between different ConfD components, e.g. transactions may not span the data model change, and all components must update any related data, while still being able to revert to the original data model in case problems are detected.

The following four sections describe the phases and corresponding MAAPI function calls that comprise this upgrade procedure, and the steps that must be taken by the program controlling the procedure. A complete example showing the procedure can be found in `examples.confd/in_service_upgrade/simple` in the bundled examples collection. The code excerpts and description refer to this example. See also the confd_lib_maapi(3) manual page for the definitions of the MAAPI functions.

13.2. Preparing for the Upgrade

All the new .fxs files, clispecs, MIBs, etc, as well as the `docroot` tree for the Web UI (if used), that are to be used after the upgrade, must be installed separately from the current ones, and the current ones may not be removed until the upgrade has completed successfully. A good way to organize this is to have the references to the installation directories in `confd.conf` use a symbolic link. This way we can switch the on-disk data to the new version by simply changing the link, without the need for complex modification of `confd.conf`. Files that are unchanged between the two versions should be duplicated, or possibly (hard-)linked if disk space is limited.

In the example, we use two directories `pkg/v1` and `pkg/v2` for the old and new versions, respectively, and a symlink `pkg/current` that points to the currently used version. In `confd.conf` both `/confdConfig/loadPath/dir` and `/confdConfig/webui/docroot` are then given with the use of the symlink. Multiple loadPath directories are of course also possible, by having subdirectories below `v1` and/or `v2`.

For MIB (.bin) files other than the ones built-in to ConfD, `confd.conf` offers two possibilities: we can either specify the actual file names with `/confdConfig/snmpAgent/mibs/file` elements, or use `/confdConfig/snmpAgent/mibs/fromLoadPath` to tell ConfD to load these files from the directories given via `/confdConfig/loadPath`. To have the symlink scheme work for the MIB files on upgrade, we need to use the latter alternative, and only specify built-in MIBs via `/confdConfig/snmpAgent/mibs/file`.

The `upgrade.c` program in the example controls the upgrade procedure. It can be run either standalone or via a `osCommand` specification in the clispec. In both cases it must connect a MAAPI socket and associate it with a user session. When running from the CLI, it must use the user session id provided by the environment variable `CONFD_MAAPI_USID` for this, otherwise it can start a new user session:

```
static int set_usess(int sock)
{
```

```
    char *user = "admin";
    const char *groups[] = {"admin"};
    char *context = "system";
    struct confd_ip ip;

    /* must use usid from CLI to be allowed to run across upgrade */
    if ((usid_env = getenv("CONFD_MAAPI_USID")) != NULL) {
        return maapi_set_user_session(sock, atoi(usid_env));
    } else {
        ip.af = AF_INET;
        inet_pton(AF_INET, "127.0.0.1", &ip.ip.v4);
        return maapi_start_user_session(sock, user, context, groups, 1,
                                        &ip, CONFD_PROTO_TCP);
    }
}
```

Applications connected to ConfD, e.g. data providers and CDB subscribers, are not directly affected by the upgrade procedure. If they need to take some action due to the upgrade, they should be subscribed to `CONFD_NOTIF_UPGRADE_EVENT` event notifications (see Chapter 12, *Notifications*), and will then be notified of the different phases of the upgrade. If nothing else, most applications should call `confd_load_schemas()` (see confd_lib_lib(3)) when an upgrade has completed, in order to update the in-memory representation of the data model. The `cdb_subscriber.c` program in the example shows how this can be done.

13.3. Initializing the Upgrade

After having set up the MAAPI socket, the first step in the actual upgrade procedure is to call the `maapi_init_upgrade()` function. Its purpose is to bring ConfD into "upgrade mode", where no transactions are running, and the northbound agents have entered a state that does not allow new transactions to be started.

```
    progress("Initializing upgrade...\n");
    phase = "Init";
    /* run notifier in separate process
       - maapi_init_upgrade() blocks */
    notifier = run_notifier(timeout, force);
    OK(maapi_init_upgrade(maapisock, timeout,
                          force ? MAAPI_UPGRADE_KILL_ON_TIMEOUT : 0));
    if (notifier != -1)
        kill(notifier, SIGTERM);
    notifier = -1;
    progress("Init OK\n");
    maapi_prio_message(maapisock, "all",
                       "\n>>> System upgrade in progress...\n");
```

If users have sessions in configure mode when this function is called, they are given the opportunity to exit from configure mode voluntarily. The function call will block until all transactions have been terminated (although not longer than specified by the timeout). For this reason, we fork() a process that periodically sends out messages to all users, informing them of the imminent upgrade.

If any transactions remain when the timeout expires, `maapi_init_upgrade()` will fail with `confd_errno` `CONFD_ERR_TIMEOUT`, unless the `MAAPI_UPGRADE_KILL_ON_TIMEOUT` flag was used. If `upgrade.c` was given the `-f` (force) option, it will pass this flag to `maapi_init_upgrade()`, and any remaining transactions will be forcibly terminated instead.

When `maapi_init_upgrade()` is called, a `CONFD_UPGRADE_INIT_STARTED` event notification is sent, and when it completes successfully, a `CONFD_UPGRADE_INIT_SUCCEEDED` event notification is sent.

Note

If the function fails, i.e. it does not return `CONFD_OK`, ConfD will automatically abort the upgrade, reverting to the pre-upgrade state, and send a `CONFD_UPGRADE_ABORTED` event notification. This is true also for the functions described in in the next two sections.

13.4. Performing the Upgrade

When `maapi_init_upgrade()` has completed successfully, the next step is to call `maapi_perform_upgrade()`. This tells ConfD to load the new .fxs files etc, and we must pass it a list of directories to load these files from. These are the directories that will become the new loadPath directories once the upgrade is complete. These directories will also be searched for CDB "init files" (see Section 5.8, “Loading initial data into CDB”), corresponding to the `/confdConfig/cdb/initPath` directories that can be specified in `confd.conf`.

```
    progress("Performing upgrade...\n");
    phase = "Perform";
    /* set up new loadpath directory */
    snprintf(buf, sizeof(buf), PKG_DIR "/%s", version);
    load_dir[0] = &buf[0];
    OK(maapi_perform_upgrade(maapisock, &load_dir[0], ndirs));
    progress("Perform OK\n");
```

At this point `confd.conf` and hence the `current` symlink must still point to the current version, and thus we pass the new directory "explicitly" to the function as `"./pkg/v2"`. In this example ConfD was also started with the `--addloadpath` option specifying an additional loadPath directory. The contents of this directory (`$CONFD_DIR/etc/confd`) does not change in the upgrade, but we must pass the same directory to `maapi_perform_upgrade()` too - the files found in the given directories will completely replace what ConfD is currently using.

A number of different problems may be detected during the loading of the new files, e.g. .fxs files may have a version that is incompatible with the ConfD version, or they may reference namespaces that can not be found. These problems will make `maapi_perform_upgrade()` fail with `confd_errno` `CONFD_ERR_BAD_CONFIG`, and `confd_lasterr()` giving information about the details of the problem. If the loading is successful, CDB will start its special upgrade transaction, and perform any automatic upgrade operations that are needed, before `maapi_perform_upgrade()` returns.

When `maapi_perform_upgrade()` completes successfully, a `CONFD_UPGRADE_PERFORMED` event notification is sent.

13.5. Committing the Upgrade

When `maapi_perform_upgrade()` has completed successfully, we must call `maapi_commit_upgrade()` to tell ConfD to make the upgrade permanent. This will also tell CDB to commit its upgrade transaction, and we may need to take some actions for this before the call:

- If the upgrade requires that an external program modifies some CDB data, it must be done at this point, using `maapi_attach_init()` as described in the CDB chapter.
- If the upgrade includes new validation points, or the validation logic for existing validation points has changed, the new validators must connect to ConfD and register for their validation points before `maapi_commit_upgrade()` is called.

In the example, all the changes can be handled by the automatic CDB upgrade, and we just proceed with the call:

```
    progress("Committing upgrade...\n");
    phase = "Commit";
    OK(maapi_commit_upgrade(maapisock));
    relink(version);
    progress("Commit OK\n");

    maapi_prio_message(maapisock, "all",
                       ">>> System upgrade has completed successfully.\n");
```

`maapi_commit_upgrade()` may fail if the upgraded data does not pass validation, and the errors returned in this case are the same as for e.g. `maapi_apply_trans()`. Since this will also make ConfD automatically revert to the pre-upgrade state, we must not change the on-disk data to reflect the upgrade until `maapi_commit_upgrade()` has succeeded. In the code above, the `relink()` call changes the symlink to point to the new version in an atomic manner.

When `maapi_commit_upgrade()` completes successfully, a `CONFD_UPGRADE_COMMITED` event notification is sent.

13.6. Aborting the Upgrade

We can abort the upgrade at any point before the `maapi_commit_upgrade()` call by calling `maapi_abort_upgrade()`. However as noted above, this should not be done when one of the other functions fails, since ConfD aborts the upgrade automatically in those cases.

When `maapi_abort_upgrade()` aborts an upgrade, a `CONFD_UPGRADE_ABORTED` event notification is sent.

13.7. Upgrade and HA

When we use the ConfD High Availability functionality, it is critical that all nodes in the HA cluster agree on the data model used. For this reason we can not do in-service upgrade on a ConfD instance that is part of a HA cluster. If ConfD is a HA slave, or a HA master with connected slaves, `maapi_init_upgrade()` will fail with `confd_errno` `CONFD_ERR_HA_WITH_UPGRADE`. Conversely, when an in-service upgrade is in progress, calling `confd_ha_beslave()` will also result in this error, and connections from slaves will be rejected.

To do the in-service upgrade on a HA cluster, we must thus use "rolling upgrade":

1. Disconnect one of the slaves from the cluster by calling `confd_ha_benone()`.
2. Upgrade the disconnected slave as described above.
3. Tell the upgraded slave to become master by calling `confd_ha_bemaster()`.
4. Upgrade the remaining nodes in the cluster one by one, telling each to connect as slave to the upgraded master by calling `confd_ha_beslave()` when the upgrade is done.

Alternatively, since the HA configuration should be able to handle that a node is stopped and restarted without service interruption, we may simply use the upgrade method described in the CDB chapter for the "rolling upgrade".

Chapter 14. The AAA infrastructure

14.1. The problem

This chapter describes how to use ConfD 's built-in authentication and authorization mechanisms. Users log into ConfD through the CLI, NETCONF, SNMP, or via the Web UI. In either case, users need to be *authenticated*. That is, a user needs to present credentials, such as a password or a public key in order to gain access.

Once a user is authenticated, all operations performed by that user need to be *authorized*. That is, certain users may be allowed to perform certain tasks, whereas others are not. This is called *authorization*. We differentiate between authorization of commands and authorization of data access.

14.2. Structure - data models

The ConfD daemon manages device configuration including *AAA* information. In fact, ConfD both manages AAA information and uses it. The AAA information describes which users may login, what passwords they have and what they are allowed to do.

This is solved in ConfD by requiring a data model to be both loaded and populated with data. ConfD uses the YANG module `tailf-aaa.yang` for authentication, while `ietf-netconf-acm.yang` (NACM, RFC 6536) as augmented by `tailf-acm.yang` is used for group assignment and authorization. For backwards compatibility, it is alternatively possible to use the older revision 2011-09-22 of `tailf-aaa.yang` for all of authentication, group assignment, and authorization. This legacy version of tailf-aaa can be found in the `$CONFD_DIR/src/confd/aaa` directory, but its usage is not further described here. Detailed information about this can be found in versions of this document for ConfD-5.3 and earlier.

14.2.1. Data model contents

The NACM data model is targeted specifically towards access control for NETCONF operations, and thus lacks some functionality that is needed in ConfD, in particular support for authorization of CLI commands and the possibility to specify the "context" (NETCONF/CLI/etc) that a given authorization rule should apply to. This functionality is modeled by augmentation of the NACM model, as defined in the `tailf-acm.yang` YANG module.

The `ietf-netconf-acm.yang` and `tailf-acm.yang` modules can be found in `$CONFD_DIR/src/confd/yang` directory in the release, while `tailf-aaa.yang` can be found in the `$CONFD_DIR/src/confd/aaa` directory.

The complete AAA data model defines a set of users, a set of groups and a set of rules. The data model must be populated with data that is subsequently used by by ConfD itself when it authenticates users and authorizes user data access. These YANG modules work exactly like all other fxs files loaded into the system with the exception that ConfD itself uses them. The data belongs to the application, but ConfD itself is the user of the data.

Since ConfD requires a data model for the AAA information for its operation, it will report an error and fail to start if these data models can not be found.

14.3. AAA related items in confd.conf

ConfD itself is configured through a configuration file - `confd.conf` . In that file we have the following items related to authentication and authorization:

`/confdConfig/aaa/sshServerKeyDir`

If SSH termination is enabled for NETCONF or the CLI, the ConfD built-in SSH server needs to have server keys. These keys are generated by the ConfD install script and by default end up in `$CONFD_DIR/etc/confd/ssh` .

It is also possible to use OpenSSH to terminate NETCONF or the CLI. If OpenSSH is used to terminate SSH traffic, the SSH keys are not necessary.

`/confdConfig/aaa/sshPubkeyAuthentication`

If SSH termination is enabled for NETCONF or the CLI, this item controls how the ConfD SSH daemon locates the user keys for public key authentication. See Section 14.4.1, “Public Key Login” for the details.

`/confdConfig/aaa/localAuthentication/enabled`

The term "local user" refers to a user stored under `/aaa/authentication/users`. The alternative is a user unknown to ConfD , typically authenticated by PAM.

By default, ConfD first checks local users before trying PAM or external authentication.

Local authentication is practical in test environments. It is also useful when we want to have one set of users that are allowed to login to the host with normal shell access and another set of users that are only allowed to access the system using the normal encrypted, fully authenticated, northbound interfaces of ConfD .

If we always authenticate users through PAM it may make sense to set this configurable to `false`. If we disable local authentication it implicitly means that we must use either PAM authentication or "external authentication". It also means that we can leave the entire data trees under `/aaa/authentication/users` and, in the case of "external auth" also `/nacm/groups` (for NACM) or `/aaa/authentication/groups` (for legacy tailf-aaa) empty.

`/confdConfig/aaa/pam`

ConfD can authenticate users using PAM (Pluggable Authentication Modules). PAM is an integral part of most Unix-like systems.

PAM is a complicated - albeit powerful - subsystem. It may be easier to have all users stored locally on the host, However if we want to store users in a central location, PAM can be used to access the remote information. PAM can be configured to perform most login scenarios including RADIUS and LDAP. One major drawback with PAM authentication is that there is no easy way to extract the group information from PAM. PAM authenticates users, it does not also assign a user to a set of groups.

PAM authentication is thoroughly described later in this chapter.

`/confdConfig/aaa/defaultGroup`	If this configuration parameter is defined and if the group of a user cannot be determined, a logged in user ends up in the given default group.
`/confdConfig/aaa/aaaBridge`	This key will be described in the Section 14.9, “Populating AAA using external data” section.
`/confdConfig/aaa/externalAuthentication`	ConfD can authenticate users using an external executable. This is further described later in the Section 14.4.4, “External authentication” section.
`/confdConfig/aaa/authenticationCallback/enabled`	If this is set to "true", ConfD will, as the last step of every authentication attempt, invoke an application callback. The callback can reject an otherwise successful authentication. See the section called “AUTHENTICATION CALLBACK” in the confd_lib_dp(3) manual page for the details about this.
`/confdConfig/aaa/authorization/callback/enabled`	If this is set to "true", ConfD will invoke application callbacks for authorization. The callbacks can partially or completely replace the logic described in Section 14.6, “Authorization”. See the section called “AUTHORIZATION CALLBACKS” in the confd_lib_dp(3) manual page for the details about this.

14.4. Authentication

Depending on northbound management protocol, when a user session is created in ConfD, it may or may not be authenticated. If the session is not yet authenticated, ConfD's AAA subsystem is used to perform authentication and authorization, as described below. If the session already has been authenticated, ConfD's AAA assigns groups to the user as described in Section 14.5, “Group Membership”, and performs authorization, as described in Section 14.6, “Authorization”.

The authentication part of the data model can be found in tailf-aaa.yang:

```
  container authentication {
    tailf:info "User management";
    container users {
      tailf:info "List of local users";
      list user {
        key name;
        leaf name {
          type string;
          tailf:info "Login name of the user";
        }
        leaf uid {
          type int32;
          mandatory true;
          tailf:info "User Identifier";
        }
        leaf gid {
          type int32;
          mandatory true;
          tailf:info "Group Identifier";
        }
        leaf password {
          type passwdStr;
```

```
          mandatory true;
        }
        leaf ssh_keydir {
          type string;
          mandatory true;
          tailf:info "Absolute path to directory where user's ssh keys
                      may be found";
        }
        leaf homedir {
          type string;
          mandatory true;
          tailf:info "Absolute path to user's home directory";
        }
      }
    }
  }
```

AAA authentication is used in the following cases:

- When the built-in SSH server is used for NETCONF and CLI sessions.
- For Web UI sessions and REST access.
- When the function `maapi_authenticate()` is used.

The different authentication mechanisms that may be used in these cases are described below. Regardless of which mechanism that is used, ConfD can optionally invoke an application callback as the last step of the authentication process, see the section called “AUTHENTICATION CALLBACK” in confd_lib_dp(3). The callback is not used for the actual authentication, but it can reject an otherwise successful authentication.

ConfD 's AAA authentication is *not* used in the following cases:

- When NETCONF uses an external SSH daemon, such as OpenSSH.

  In this case, the NETCONF session is initiated using the program **netconf-subsys**, as described in Section 15.3, “NETCONF Transport Protocols”.

- When NETCONF uses TCP, as described in Section 15.3, “NETCONF Transport Protocols”, e.g. through the command **netconf-console**.
- When the CLI uses an external SSH daemon, such as OpenSSH, or a telnet daemon.

  In this case, the CLI session is initiated through the command **confd_cli** . An important special case here is when a user has logged in to the host and invokes the command **confd_cli** from the shell.

- When SNMP is used. SNMP has its own authentication mechanisms. See Section 17.5.2, “USM and VACM and ConfD AAA” .
- When the function `maapi_start_user_session()` is used without a preceding call of `maapi_authenticate()`.

14.4.1. Public Key Login

When a user logs in over NETCONF or the CLI using the built-in SSH server, with public key login, the procedure is as follows.

The user presents a username in accordance with the SSH protocol. The SSH server consults the settings for `/confdConfig/aaa/sshPubkeyAuthentication` and `/confdConfig/aaa/localAuthentication/enabled` .

1. If `sshPubkeyAuthentication` is set to `local`, and the SSH keys in `/aaa/authentication/users/user{$USER}/ssh_keydir` match the keys presented by the user, authentication succeeds.

2. Otherwise, if `sshPubkeyAuthentication` is set to `system`, `localAuthentication` is enabled, and the SSH keys in `/aaa/authentication/users/user{$USER}/ssh_keydir` match the keys presented by the user, authentication succeeds.

3. Otherwise, if `sshPubkeyAuthentication` is set to `system` and the user `/aaa/authentication/users/user{$USER}` does not exist, but the user does exist in the OS password database, the keys in the user's `$HOME/.ssh` directory are checked. If these keys match the keys presented by the user, authentication succeeds.

4. Otherwise, authentication fails.

In all cases the keys are expected to be stored in a file called `authorized_keys` (or `authorized_keys2` if `authorized_keys` does not exist), and in the native OpenSSH format (i.e. as generated by the OpenSSH **ssh-keygen** command). If authentication succeeds, the user's group membership is established as described in Section 14.5, “Group Membership”.

This is exactly the same procedure that is used by the OpenSSH server with the exception that the built-in SSH server also may locate the directory containing the public keys for a specific user by consulting the `/aaa/authentication/users` tree.

Setting up Public Key Login

We need to provide a directory where SSH keys are kept for a specific user, and give the absolute path to this directory for the `/aaa/authentication/users/user/ssh_keydir` leaf. If public key login is not desired at all for a user, the value of the `ssh_keydir` leaf should be set to `""`, i.e. the empty string. Similarly, if the directory does not contain any SSH keys, public key logins for that user will be disabled.

The built-in SSH daemon supports DSA and RSA keys. To generate and enable RSA keys of size 4096 bits for, say, user "bob", the following steps are required.

On the client machine, as user "bob", generate a private/public key pair as:

```
# ssh-keygen -b 4096 -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/bob/.ssh/id_rsa):
Created directory '/home/bob/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/bob/.ssh/id_rsa.
Your public key has been saved in /home/bob/.ssh/id_rsa.pub.
The key fingerprint is:
ce:1b:63:0a:f9:d4:1d:04:7a:1d:98:0c:99:66:57:65 bob@buzz
# ls -lt ~/.ssh
total 8
-rw-------  1 bob users 3247 Apr  4 12:28 id_rsa
-rw-r--r--  1 bob users  738 Apr  4 12:28 id_rsa.pub
```

Now we need to copy the public key to the target machine where the NETCONF or CLI SSH client runs.

Assume we have the following user entry:

```
<user>
  <name>bob</name>
  <uid>100</uid>
  <gid>10</gid>
  <password>$1$feedbabe$nGlMYlZpQ0bzenyFOQI3L1</password>
  <ssh_keydir>/var/system/users/bob/.ssh</ssh_keydir>
  <homedir>/var/system/users/bob</homedir>
</user>
```

We need to copy the newly generated file `id_rsa.pub`, which is the public key, to a file on the target machine called `/var/system/users/bob/.ssh/authorized_keys`

14.4.2. Password Login

Password login is triggered in the following cases:

- When a user logs in over NETCONF or the CLI using the built in SSH server, with a password. The user presents a username and a password in accordance with the SSH protocol.
- When a user logs in using the Web UI. The Web UI asks for a username and password.
- When the function `maapi_authenticate()` is used.

In this case, ConfD will by default try local authentication, PAM, and external authentication, in that order, as described below. It is possible to change the order in which these are tried, by modifying the `confd.conf` . parameter `/confdConfig/aaa/authOrder` . See confd.conf(5) for details.

1. If `/aaa/authentication/users/user{$USER}` exists and the presented password matches the encrypted password in `/aaa/authentication/users/user{$USER}/password` the user is authenticated.
2. If the password does not match or if the user does not exist in `/aaa/authentication/users`, PAM login is attempted, if enabled. See ???? for details.
3. If all of the above fails and external authentication is enabled, the configured executable is invoked. See ???? for details.

If authentication succeeds, the user's group membership is established as described in ????.

14.4.3. PAM

On operating systems supporting PAM, ConfD also supports PAM authentication. Using PAM authentication with ConfD can be very convenient since it allows us to have the same set of users and groups having access to ConfD as those that have access to the UNIX/Linux host itself.

If we use PAM, we do not have to have any users or any groups configured in the ConfD aaa namespace at all. To configure PAM we typically need to do the following:

1. Remove all users and groups from the aaa initialization XML file.
2. Enable PAM in confd.conf by adding:

```
<pam>
  <enabled>true</enabled>
  <service>common-auth</service>
</pam>
```

to the aaa section in confd.conf . The `service` name specifies the PAM service, typically a file in the directory `/etc/pam.d`, but may alternatively be an entry in a file `/etc/pam.conf`, depending on OS and version. Thus it is possible to have a different login procedure to ConfD than to the host itself.

3. If pam is enabled and we want to use pam for login the system may have to run as root. This depends on how pam is configured locally. However the default "system-auth" will typically require root since the pam libraries then read `/etc/shadow`. If we don't want to run ConfD as root, the solution here is to change owner of a helper program called `$CONFD_DIR/lib/confd/lib/core/pam/priv/epam` and also set the setuid bit.

```
# cd $CONFD_DIR/lib/confd/lib/core/pam/priv/
# chown root:root epam
# chmod u+s epam
```

PAM is the recommended way to authenticate ConfD users.

As an example, say that we have user test in `/etc/passwd`, and furthermore:

```
# grep test /etc/group
operator:x:37:test
admin:x:1001:test
```

thus, the test user is part of the admin and the operator groups and logging in to ConfD as the test user, through CLI ssh, Web UI, or netconf renders the following in the audit log.

```
<INFO> 28-Jan-2009::16:05:55.663 buzz confd[14658]: audit user: test/0 logged
    in over ssh from 127.0.0.1 with authmeth:password
<INFO> 28-Jan-2009::16:05:55.670 buzz confd[14658]: audit user: test/5 assigned
    to groups: operator,admin
<INFO> 28-Jan-2009::16:05:57.655 buzz confd[14658]: audit user: test/5 CLI 'exit'
```

Thus, the test user was found and authenticated from `/etc/passwd`, and the crucial group assignment of the test user was done from `/etc/group`.

If we wish to be able to also manipulate the users, their passwords etc on the device we can write a private YANG model for that data, store that data in CDB, setup a normal CDB subscriber for that data, and finally when our private user data is manipulated, our CDB subscriber picks up the changes and changes the contents of the relevant `/etc` files.

14.4.4. External authentication

A common situation is when we wish to have all authentication data stored remotely, not locally, for example on a remote RADIUS or LDAP server. This remote authentication server typically not only stores the users and their passwords, but also the group information.

If we wish to have not only the users, but also the group information stored on a remote server, the best option for ConfD authentication is to use "external authentication".

If this feature is configured, ConfD will invoke the executable configured in `/confdConfig/aaa/externalAuthentication/executable` in `confd.conf` , and pass the username and the clear text password on `stdin` using the string notation: `"[user;password;]\n"`.

For example if user "bob" attempts to login over SSH using the password "secret", and external authentication is enabled, ConfD will invoke the configured executable and write `"[bob;secret;]\n"` on the `stdin` stream for the executable.

The task of the executable is then to authenticate the user and also establish the username-to-groups mapping.

For example the executable could be a RADIUS client which utilizes some proprietary vendor attributes to retrieve the groups of the user from the RADIUS server. If authentication is successful, the program should write `"accept "` followed by a space-separated list of groups the user is member of, and additional information as described below. Again, assuming that Bob's password indeed was "secret", and that Bob is member of the "admin" and the "lamers" groups, the program should write `"accept admin lamers $uid $gid $supplementary_gids $HOME\n"` on its standard output and then exit.

Thus the format of the output from an "externalauth" program when authentication is successful should be:

```
"accept $groups $uid $gid $supplementary_gids $HOME\n"
```

Where

- `$groups` is a space separated list of the group names the user is a member of.
- `$uid` is the UNIX integer user id ConfD should use as default when executing commands for this user.
- `$gid` is the UNIX integer group id ConfD should use as default when executing commands for this user.
- `$supplementary_gids` is a (possibly empty) space separated list of additional UNIX group ids the user is also a member of.
- `$HOME` is the directory which should be used as HOME for this user when ConfD executes commands on behalf of this user.

It is also possible for the program to return additional information on successful authentication, by using `"accept_info"` instead of `"accept"`:

```
"accept_info $groups $uid $gid $supplementary_gids $HOME $info\n"
```

Where `$info` is some arbitrary text. ConfD will then just append this text to the generated audit log message (CONFD_EXT_LOGIN).

If authentication failed, the program should write `"reject"` or `"abort"`, possibly followed by a reason for the rejection, and a trailing newline. For example `"reject Bad password\n"` or just `"abort \n"`. The difference between `"reject"` and `"abort"` is that with `"reject"`, ConfD will try subsequent mechanisms configured for `/confdConfig/aaa/authOrder` in `confd.conf` (if any), while with `"abort"`, the authentication fails immediately. Thus `"abort"` can prevent subsequent mechanisms from being tried, but when external authentication is the last mechanism (as in the default order), it has the same effect as `"reject"`.

When external authentication is used, the group list returned by the external program is prepended by any possible group information stored locally under the `/aaa` tree. Hence when we use external authentication

it is indeed possible to have the entire `/aaa/authentication` tree empty. The group assignment performed by the external program will still be valid and the relevant groups will be used by ConfD when the authorization rules are checked.

14.5. Group Membership

Once a user is authenticated, group membership must be established. A single user can be a member of several groups. Group membership is used by the authorization rules to decide which operations a certain user is allowed to perform. Thus the ConfD AAA authorization model is entirely group based. This is also sometimes referred to as role based authorization.

All groups are stored under `/nacm/groups`, and each group contains a number of usernames. The `ietf-netconf-acm.yang` model defines a group entry:

```
list group {
  key name;

  description
    "One NACM Group Entry.  This list will only contain
     configured entries, not any entries learned from
     any transport protocols.";

  leaf name {
    type group-name-type;
    description
      "Group name associated with this entry.";
  }

  leaf-list user-name {
    type user-name-type;
    description
      "Each entry identifies the username of
       a member of the group associated with
       this entry.";
  }
}
```

The `tailf-acm.yang` model augments this with a `gid` leaf:

```
augment /nacm:nacm/nacm:groups/nacm:group {
  leaf gid {
    type int32;
    description
      "This leaf associates a numerical group ID with the group.
       When a OS command is executed on behalf of a user,
       supplementary group IDs are assigned based on 'gid' values
       for the groups that the use is a member of.";
  }
}
```

A valid group entry could thus look like:

```
<group>
  <name>admin</name>
  <user-name>bob</user-name>
  <user-name>joe</user-name>
```

```
  <gid xmlns="http://tail-f.com/yang/acm">99</gid>
</group>
```

The above XML data would then mean that users `bob` and `joe` are members of the `admin` group. The users need not necessarily exist as actual users under `/aaa/authentication/users` in order to belong to a group. If for example PAM authentication is used, it does not make sense to have all users listed under `/aaa/authentication/users`.

By default, the user is assigned to groups by using any groups provided by the northbound transport (e.g. via the **confd_cli** or **netconf-subsys** programs), by consulting data under `/nacm/groups`, by consulting the `/etc/group` file, and by using any additional groups supplied by the authentication method. If `/nacm/enable-external-groups` is set to "false", only the data under `/nacm/groups` is consulted.

The resulting group assignment is the union of these methods, if it is non-empty. Otherwise, the default group is used, if configured (`/confdConfig/aaa/defaultGroup` in `confd.conf`).

A user entry has a UNIX uid and UNIX gid assigned to it. Groups may have optional group ids. When a user is logged in, and ConfD tries to execute commands on behalf of that user, the uid/gid for the command execution is taken from the user entry. Furthermore, UNIX supplementary group ids are assigned according to the gids in the groups where the user is a member.

14.6. Authorization

Once a user is authenticated and group membership is established, when the user starts to perform various actions, each action must be authorized. Normally the authorization is done based on rules configured in the AAA data model as described in this section, but if needed we can also register application callbacks to partially or completely replace this logic, see the section called “AUTHORIZATION CALLBACKS” in confd_lib_dp(3).

The authorization procedure first checks the value of `/nacm/enable-nacm`. This leaf has a default of `true`, but if it is set to `false`, all access is permitted. Otherwise, the next step is to traverse the `rule-list` list:

```
list rule-list {
  key "name";
  ordered-by user;
  description
    "An ordered collection of access control rules.";

  leaf name {
    type string {
      length "1..max";
    }
    description
      "Arbitrary name assigned to the rule-list.";
  }
  leaf-list group {
    type union {
      type matchall-string-type;
      type group-name-type;
    }
    description
      "List of administrative groups that will be
       assigned the associated access rights
       defined by the 'rule' list.
```

```
      The string '*' indicates that all groups apply to the
      entry.";
  }

  // ...
}
```

If the `group` leaf-list in a `rule-list` entry matches any of the user's groups, the `cmdrule` list entries are examined for command authorization, while the `rule` entries are examined for rpc, notification, and data authorization.

14.6.1. Command authorization

The `tailf-acm.yang` module augments the `rule-list` entry in `ietf-netconf-acm.yang` with a `cmdrule` list:

```
augment /nacm:nacm/nacm:rule-list {

  list cmdrule {
    key "name";
    ordered-by user;
    description
      "One command access control rule. Command rules control access
       to CLI commands and Web UI functions.

       Rules are processed in user-defined order until a match is
       found.  A rule matches if 'context', 'command', and
       'access-operations' match the request.  If a rule
       matches, the 'action' leaf determines if access is granted
       or not.";

    leaf name {
      type string {
        length "1..max";
      }
      description
        "Arbitrary name assigned to the rule.";
    }

    leaf context {
      type union {
        type nacm:matchall-string-type;
        type string;
      }
      default "*";
      description
        "This leaf matches if it has the value '*' or if its value
         identifies the agent that is requesting access, i.e. 'cli'
         for CLI or 'webui' for Web UI.";
    }

    leaf command {
      type string;
      default "*";
      description
        "Space-separated tokens representing the command. Refer
         to the Tail-f AAA documentation for further details.";
```

```
    }

    leaf access-operations {
      type union {
        type nacm:matchall-string-type;
        type nacm:access-operations-type;
      }
      default "*";
      description
        "Access operations associated with this rule.

         This leaf matches if it has the value '*' or if the
         bit corresponding to the requested operation is set.";
    }

    leaf action {
      type nacm:action-type;
      mandatory true;
      description
        "The access control action associated with the
         rule.  If a rule is determined to match a
         particular request, then this object is used
         to determine whether to permit or deny the
         request.";
    }

    leaf log-if-permit {
      type empty;
      description
        "If this leaf is present, access granted due to this rule
         is logged in the developer log. Otherwise, only denied
         access is logged. Mainly intended for debugging of rules.";
    }

    leaf comment {
      type string;
      description
        "A textual description of the access rule.";
    }
  }
}
```

Each rule has seven leafs. The first is the `name` list key, the following three leafs are matching leafs. When ConfD tries to run a command it tries to match the command towards the matching leafs and if all of `context`, `command`, and `access-operations` match, the fifth field, i.e. the `action`, is applied.

name — `name` is the name of the rule. The rules are checked in order, with the ordering given by the the YANG `ordered-by user` semantics, i.e. independent of the key values.

context — `context` is either of the strings `cli`, `webui`, or `*` for a command rule. This means that we can differentiate authorization rules for which access method is used. Thus if command access is attempted through the CLI the context will be the string `cli` whereas for operations via the Web UI, the context will be the string `webui`.

command — This is the actual command getting executed. If the rule applies to one or several CLI commands, the string is a space separated list of CLI command tokens, for example `request system reboot`. If the command applies

to Web UI operations, it is a space separated string similar to a CLI string. A string which consists of just "*" matches any command.

It is important to understand that a command rule for the CLI applies to the string as entered by the user. The command rules are not aware of the data model. Thus it is not possible to have a rule like:

```
<cmdrule>
  <name>delete-eth0</name>
  <context>cli</context>
  <command>delete interfaces interface eth0</command>
  <access-operations>exec</access-operations>
  <action>deny</action>
</cmdrule>
```

to protect a specific interface from removal in The Juniper CLI. The user can enter:

```
joe@host% edit interfaces
joe@host% delete interface eth0
```

making the command rule above moot.

In the Cisco like CLIs it makes more sense to use command rules to protect data. This is due to the command oriented character of the Cisco CLIs.

In general, we do not recommend using command rules to protect the configuration. Use rules for data access as described in the next section to control access to different parts of the data. Command rules should be used only for CLI commands and Web UI operations that cannot be expressed as data rules.

Another thing that is important for command rule processing of CLI commands is the mode. If we enable the feature `/confdConfig/cli/modeInfoInAAA`, the command rule matching will match on a string where the CLI mode is prepended. This makes command rule processing more useful in the Cisco style CLIs than in the Juniper style CLI.

The individual tokens can be POSIX extended regular expressions. Each regular expression is implicitly anchored, i.e. an "^" is prepended and a "$" is appended to the regular expression.

access-operations

`access-operations` is used to match the operation that ConfD tries to perform. It must be one or both of the "read" and "exec" values from the `access-operations-type` bits type definition in `ietf-netconf-acm.yang`, or "*" to match any operation.

action

If all of the previous fields match, the rule as a whole matches and the value of `action` will be taken. I.e. if a match is found, a decision is made whether to permit or deny the request in its entirety. If `action` is `permit`, the request is permitted, if `action` is `deny`, the request is denied and an entry written to the developer log.

log-if-permit

If this leaf is present, an entry is written to the developer log for a matching request also when `action` is `permit`. This is very useful when debugging command rules.

comment An optional textual description of the rule.

For the rule processing to be written to the devel log, the `/confdConfig/logs/developer-LogLevel` entry in `confd.conf` must be set to `trace`.

If no matching rule is found in any of the `cmdrule` lists in any `rule-list` entry that matches the user's groups, this augmentation from `tailf-acm.yang` is relevant:

```
augment /nacm:nacm {
  leaf cmd-read-default {
    type nacm:action-type;
    default "permit";
    description
      "Controls whether command read access is granted
       if no appropriate cmdrule is found for a
       particular command read request.";
  }

  leaf cmd-exec-default {
    type nacm:action-type;
    default "permit";
    description
      "Controls whether command exec access is granted
       if no appropriate cmdrule is found for a
       particular command exec request.";
  }

  leaf log-if-default-permit {
    type empty;
    description
      "If this leaf is present, access granted due to one of
       /nacm/read-default, /nacm/write-default, or /nacm/exec-default
       /nacm/cmd-read-default, or /nacm/cmd-exec-default
       being set to 'permit' is logged in the developer log.
       Otherwise, only denied access is logged. Mainly intended
       for debugging of rules.";
  }
}
```

- If "read" access is requested, the value of `/nacm/cmd-read-default` determines whether access is permitted or denied.
- If "exec" access is requested, the value of `/nacm/cmd-exec-default` determines whether access is permitted or denied.

If access is permitted due to one of these default leafs, the `/nacm/log-if-default-permit`has the same effect as the `log-if-permit` leaf for the `cmdrule` lists.

14.6.2. Rpc, notification, and data authorization

The rules in the `rule` list are used to control access to rpc operations, notifications, and data nodes defined in YANG models. Access to invocation of actions (`tailf:action`) is controlled with the same method as access to data nodes, with a request for "exec" access. `ietf-netconf-acm.yang` defines a `rule` entry as:

```
list rule {
```

```
key "name";
ordered-by user;
description
  "One access control rule.

   Rules are processed in user-defined order until a match is
   found.  A rule matches if 'module-name', 'rule-type', and
   'access-operations' match the request.  If a rule
   matches, the 'action' leaf determines if access is granted
   or not.";

leaf name {
  type string {
    length "1..max";
  }
  description
    "Arbitrary name assigned to the rule.";
}

leaf module-name {
  type union {
    type matchall-string-type;
    type string;
  }
  default "*";
  description
    "Name of the module associated with this rule.

     This leaf matches if it has the value '*' or if the
     object being accessed is defined in the module with the
     specified module name.";
}
choice rule-type {
  description
    "This choice matches if all leafs present in the rule
     match the request.  If no leafs are present, the
     choice matches all requests.";
  case protocol-operation {
    leaf rpc-name {
      type union {
        type matchall-string-type;
        type string;
      }
      description
        "This leaf matches if it has the value '*' or if
         its value equals the requested protocol operation
         name.";
    }
  }
  case notification {
    leaf notification-name {
      type union {
        type matchall-string-type;
        type string;
      }
      description
        "This leaf matches if it has the value '*' or if its
         value equals the requested notification name.";
    }
  }
```

```
    case data-node {
      leaf path {
        type node-instance-identifier;
        mandatory true;
        description
          "Data Node Instance Identifier associated with the
           data node controlled by this rule.

           Configuration data or state data instance
           identifiers start with a top-level data node.  A
           complete instance identifier is required for this
           type of path value.

           The special value '/' refers to all possible
           data-store contents.";
      }
    }
  }

  leaf access-operations {
    type union {
      type matchall-string-type;
      type access-operations-type;
    }
    default "*";
    description
      "Access operations associated with this rule.

       This leaf matches if it has the value '*' or if the
       bit corresponding to the requested operation is set.";
  }

  leaf action {
    type action-type;
    mandatory true;
    description
      "The access control action associated with the
       rule.  If a rule is determined to match a
       particular request, then this object is used
       to determine whether to permit or deny the
       request.";
  }

  leaf comment {
    type string;
    description
      "A textual description of the access rule.";
  }
}
```

`tailf-acm` augments this with two additional leafs:

```
augment /nacm:nacm/nacm:rule-list/nacm:rule {

  leaf context {
    type union {
      type nacm:matchall-string-type;
      type string;
    }
```

```
    default "*";
    description
      "This leaf matches if it has the value '*' or if its value
       identifies the agent that is requesting access, e.g. 'netconf'
       for NETCONF, 'cli' for CLI, or 'webui' for Web UI.";

  }

  leaf log-if-permit {
    type empty;
    description
      "If this leaf is present, access granted due to this rule
       is logged in the developer log. Otherwise, only denied
       access is logged. Mainly intended for debugging of rules.";
  }
}
```

Similar to the command access check, whenever a user through some agent tries to access an rpc, a notification, a data item, or an action, access is checked. For a rule to match, three or four leafs must match and when a match is found, the corresponding action is taken.

We have the following leafs in the `rule` list entry.

name
: `name` is the name of the rule. The rules are checked in order, with the ordering given by the the YANG `ordered-by user` semantics, i.e. independent of the key values.

module-name
: The `module-name` string is the name of the YANG module the rule applies to. The special value `*` matches all modules.

rpc-name / notification-name / path
: This is a choice between three possible leafs that are used for matching:

  rpc-name
  : The name of a rpc operation, or "*" to match any rpc.

  notification-name
  : The name of a notification, or "*" to match any notification.

  path
  : A restricted XPath expression leading down into the populated XML tree. A rule with a path specified matches if it is equal to or shorter than the checked path. Several types of paths are allowed.

    1. Tagpaths that are not containing any keys. For example `/interfaces/interface/mtu`.
    2. Instantiated key: as in `/interfaces/interface[name="eth0"]/mask` matches the mask element only for the interface name "eth0". It's possible to have partially instantiated paths only containing some keys instantiated - i.e combinations of tagpaths and keypaths. Assuming a deeper tree, the path `/hosts/host[name="venus"]/servers/server/ip` matches the "ip" element for all servers, but only for the host named "venus".

3. Wild card at end as in: `/interfaces/interface/*` does not match `/interfaces/interface` but rather all children of that path.

Thus the path in a rule is matched against the path in the attempted data access. If the attempted access has a path that is equal to or longer than the rule path - we have a match.

If none of the leafs `rpc-name`, `notification-name`, or `path` are set, the rule matches for any rpc, notification, data, or action access.

context — `context` is either of the strings `cli`, `netconf`, `webui`, `snmp`, or `*` for a data rule. Furthermore, when we initiate user sessions from MAAPI, we can choose any string we want.

Similarly to command rules we can differentiate access depending on which agent is used to gain access.

access-operations — `access-operations` is used to match the operation that ConfD tries to perform. It must be one or more of the "create", "read", "update", "delete" and "exec" values from the `access-operations-type` bits type definition in `ietf-netconf-acm.yang`, or "*" to match any operation.

action — This leaf has the same characteristics as the `action` leaf for command access.

log-if-permit — This leaf has the same characteristics as the `log-if-permit` leaf for command access.

comment — An optional textual description of the rule.

If no matching rule is found in any of the `rule` lists in any `rule-list` entry that matches the user's groups, the data model node for which access is requested is examined for presence of the NACM extensions:

- If the `nacm:default-deny-all` extension is specified for the data model node, access is denied.
- If the `nacm:default-deny-write` extension is specified for the data model node, and "create", "update", or "delete" access is requested, access is denied.

If examination of the NACM extensions did not result in access being denied, the value (`permit` or `deny`) of the relevant default leaf is examined:

- If "read" access is requested, the value of `/nacm/read-default` determines whether access is permitted or denied.
- If "create", "update", or "delete" access is requested, the value of `/nacm/write-default` determines whether access is permitted or denied.
- If "exec" access is requested, the value of `/nacm/exec-default` determines whether access is permitted or denied.

If access is permitted due to one of these default leafs, this augmentation from `tailf-acm.yang` is relevant:

```
augment /nacm:nacm {
  ...
  leaf log-if-default-permit {
    type empty;
    description
      "If this leaf is present, access granted due to one of
       /nacm/read-default, /nacm/write-default, /nacm/exec-default
       /nacm/cmd-read-default, or /nacm/cmd-exec-default
       being set to 'permit' is logged in the developer log.
       Otherwise, only denied access is logged. Mainly intended
       for debugging of rules.";
  }
}
```

I.e. it has the same effect as the `log-if-permit` leaf for the `rule` lists, but for the case where the value of one of the default leafs permits the access.

When ConfD executes a command, the command rules in the authorization database are searched, The rules are tried in order, as described above. When a rule matches the operation (command) that ConfD is attempting, the action of the matching rule is applied - whether permit or deny.

When actual data access is attempted, the data rules are searched. E.g. when a user attempts to execute `delete aaa` in the CLI, the user needs delete access to the entire tree `/aaa`.

Another example is if a CLI user writes `show configuration aaa TAB` it suffices to have read access to at least one item below `/aaa` for the CLI to perform the TAB completion. If no rule matches or an explicit deny rule is found, the CLI will not TAB complete.

Yet another example is if a user tries to execute `delete aaa authentication users`, we need to perform a check on the paths `/aaa` and `/aaa/authentication` before attempting to delete the sub tree. Say that we have a rule for path `/aaa/authentication/users` which is an permit rule and we have a subsequent rule for path `/aaa` which is a deny rule. With this rule set the user should indeed be allowed to delete the entire `/aaa/authentication/users` tree but not the `/aaa` tree nor the `/aaa/authentication` tree.

We have two variations on how the rules are processed. The easy case is when we actually try to read or write an item in the configuration database. The execution goes like:

```
foreach rule {
    if (match(rule, path)) {
       return rule.action;
    }
}
```

The second case is when we execute TAB completion in the CLI. This is more complicated. The execution goes like:

```
rules = select_rules_that_may_match(rules, path);
if (any_rule_is_permit(rules))
    return permit;
else
    return deny;
```

The idea being that as we traverse (through TAB) down the XML tree, as long as there is at least one rule that can possibly match later, once we have more data, we must continue.

For example assume we have:

1. `"/system/config/foo" --> permit`

2. `"/system/config" --> deny`

If we in the CLI stand at `"/system/config"` and hit TAB we want the CLI to show `foo` as a completion, but none of the other nodes that exist under `/system/config`. Whereas if we try to execute `delete /system/config` the request must be rejected.

14.6.3. Authorization Examples

Assume that we have two groups, `admin` and `oper`. We want `admin` to be able to see and and edit the XML tree rooted at `/aaa`, but we do not want users that are members of the `oper` group to even see the `/aaa` tree. We would have the following rule-list and rule entries. Note, here we use the XML data from `tailf-aaa.yang` to exemplify. The examples apply to all data, for all data models loaded into the system.

```
<rule-list>
  <name>admin</name>
  <group>admin</group>
  <rule>
    <name>tailf-aaa</name>
    <module-name>tailf-aaa</module-name>
    <path>/</path>
    <access-operations>read create update delete</access-operations>
    <action>permit</action>
  </rule>
</rule-list>
<rule-list>
  <name>oper</name>
  <group>oper</group>
  <rule>
    <name>tailf-aaa</name>
    <module-name>tailf-aaa</module-name>
    <path>/</path>
    <access-operations>read create update delete</access-operations>
    <action>deny</action>
  </rule>
</rule-list>
```

If we do not want the members of `oper` to be able to execute the NETCONF operation `edit-config`, we define the following rule-list and rule entries:

```
<rule-list>
  <name>oper</name>
  <group>oper</group>
  <rule>
    <name>edit-config</name>
    <rpc-name>edit-config</rpc-name>
    <context xmlns="http://tail-f.com/yang/acm">netconf</context>
    <access-operations>exec</access-operations>
    <action>deny</action>
  </rule>
</rule-list>
```

To spell it out, the above defines four elements to match. If ConfD tries to perform a `netconf` operation, which is the operation `edit-config`, and the user which runs the command is member of the `oper` group, and finally it is an `exec` (execute) operation, we have a match. If so, the action is `deny`.

The `path` leaf can be used to specify explicit paths into the XML tree using XPath syntax. For example the following:

```
<rule-list>
  <name>admin</name>
  <group>admin</group>
  <rule>
    <name>bob-password</name>
    <module-name>tailf-aaa</module-name>
    <path>/aaa/authentication/users/user[name='bob']/password</path>
    <context xmlns="http://tail-f.com/yang/acm">cli</context>
    <access-operations>read update</access-operations>
    <action>permit</action>
  </rule>
</rule-list>
```

Explicitly allows the `admin` group to change the password for precisely the `bob` user when the user is using the CLI. Had `path` been `/aaa/authentication/users/user/password` the rule would apply to all password elements for all users.

ConfD applies variable substitution, whereby the username of the logged in user can be used in a `path`. Thus:

```
<rule-list>
  <name>admin</name>
  <group>admin</group>
  <rule>
    <name>user-password</name>
    <module-name>tailf-aaa</module-name>
    <path>/aaa/authentication/users/user[name='$USER']/password</path>
    <context xmlns="http://tail-f.com/yang/acm">cli</context>
    <access-operations>read update</access-operations>
    <action>permit</action>
  </rule>
</rule-list>
```

The above rule allows all users that are part of the `admin` group to change their own passwords only.

Finally if we wish members of the `oper` group to never be able to execute the `request system reboot` command, also available as a `reboot` NETCONF rpc, we have:

```
<rule-list>
  <name>oper</name>
  <group>oper</group>

  <cmdrule xmlns="http://tail-f.com/yang/acm">
    <name>request-system-reboot</name>
    <context>cli</context>
    <command>request system reboot</command>
```

```
    <access-operations>exec</access-operations>
    <action>deny</action>
  </cmdrule>

  <!-- The following rule is required since the user can -->
  <!-- do "edit system" -->

  <cmdrule xmlns="http://tail-f.com/yang/acm">
    <name>request-reboot</name>
    <context>cli</context>
    <command>request reboot</command>
    <access-operations>exec</access-operations>
    <action>deny</action>
  </cmdrule>

  <rule>
    <name>netconf-reboot</name>
    <rpc-name>reboot</rpc-name>
    <context xmlns="http://tail-f.com/yang/acm">netconf</context>
    <access-operations>exec</access-operations>
    <action>deny</action>
  </rule>

</rule-list>
```

Debugging the AAA rules can be hard. The best way to debug rules that behave unexpectedly is to add the `log-if-permit` leaf to some or all of the rules that have `action permit`. Whenever such a rule triggers a permit action, an entry is written to the developer log.

Finally it is worth mentioning that when a user session is initially created it will gather the authorization rules that are relevant for that user session and keep these rules for the life of the user session. Thus when we update the AAA rules in e.g. the CLI the update will not apply to the current session - only to future user sessions.

14.7. The AAA cache

ConfD's AAA subsystem will cache the AAA information in order to speed up the authorization process. This cache must be updated whenever there is a change to the AAA information. The mechanism for this update depends on how the AAA information is stored, as described in the following two sections.

14.8. Populating AAA using CDB

In order to start ConfD , the data models for AAA must be loaded. The defaults in the case that no actual data is loaded for these models allow all read and exec access, while write access is denied. Access may still be further restricted by the NACM extensions, though - e.g. the `/nacm` container has `nacm:default-deny-all`, meaning that not even read access is allowed if no data is loaded.

The AAA data can either be stored in CDB or in an external daemon as described in the chapter Chapter 7, *The external database API*. The only new problem when we use CDB to store the AAA data is to initialize the AAA database. This can be done as described in the chapter Chapter 5, *CDB - The ConfD XML Database*, from an XML document containing real data.

ConfD ships with a decent initialization document for the AAA database. The file is called `aaa_init.xml` and is by default copied to the CDB directory by the ConfD install scripts. The file defines two users, `admin` and `oper` with passwords set to `admin` and `oper` respectively.

Normally the AAA data will be stored as configuration in CDB. This allows for changes to be made through ConfD's transaction-based configuration management. In this case the AAA cache will be updated automatically when changes are made to the AAA data. If changing the AAA data via ConfD's configuration management is not possible or desirable, it is alternatively possible to use the CDB operational data store for AAA data. In this case the AAA cache can be updated either explicitly e.g. by using the `maapi_aaa_reload()` function, see the ???? manual page, or by triggering a subscription notification by using the "subscription lock" when updating the CDB operational data store, see ????.

14.9. Populating AAA using external data

Note

The `confd_aaa_bridge` program described here is deprecated. It does not support the NACM data model. It may still be useful to study this as an example of an external data provider for AAA data, however the implementation follows the same principles as for other external data providers.

An alternative to storing the AAA data in CDB is to store it outside of ConfD. ConfD comes with an example implementation of such a program. It is called `confd_aaa_bridge` and is fully described in the man page confd_aaa_bridge(1).

The procedure here is precisely the same as with any other data model - with the exception that ConfD itself is a user of this data and reads it. Thus if using CDB to store the AAA data is not an option, the API for external databases must be used to populate the AAA tree.

The YANG model which describes this is the same `tailf-aaa.yang` but annotated with a callpoint. It is shipped together with the example implementation called `confd_aaa_bridge`. When we compile the `tailf-aaa.yang` with the callpoint for external data we name the resulting file `aaa_bridge.fxs`.

Note

The name of the fxs file is not significant, the `aaa_bridge.fxs` name is used here only to distinguish it from the `aaa_cdb.fxs` name that has been used for the CDB version of legacy tailf-aaa. We can equally well use the more "natural" name `tailf-aaa.fxs` in both cases.

Note

We must additionally annotate the `ietf-netconf-acm.yang` module with a callpoint and compile it, if the group assignment and authorization data is to be provided by an external data provider. This annotation must be done *in addition to* the annotation done by the `ietf-netconf-acm-ann.yang` module included in the ConfD release.

The example program `confd_aaa_bridge.c` which is delivered as source code in the ConfD release contains an example implementation of external storage of the AAA data in an ad hoc .ini style file. (This is only of interest for users that do not use CDB to store any data at all.) See the UNIX man page confd_aaa_bridge(1).

The `confd_aaa_bridge` program implements a configuration data provider, and thus changes to the AAA data can be made through ConfD's configuration management just as when the data is stored as configuration in CDB. And similar to the CDB case, if changing the AAA data via ConfD's configuration management is not possible or desirable, it is alternatively possible to let the AAA data be provided by an operational (i.e. read-only) external data provider. In either case, when we use an external data provider for AAA, the AAA cache must always be updated explicitly, e.g. by using the `maapi_aaa_reload()` function, see the confd_lib_maapi(3) manual page. For a configuration data

provider, the `confd_aaa_reload()` function may be more convenient, see the confd_lib_dp(3) manual page - this function is used by `confd_aaa_bridge`.

14.10. Hiding the AAA tree

Some applications may not want to expose the AAA data to end users in the CLI or the Web UI. Two reasonable approaches exist here and both rely on the `tailf:export` statement. If a module has `tailf:export none` it will be invisible to all agents. We can then either use a transform whereby we define another AAA model and write a transform program which maps our AAA data to the data which must exist in `tailf-aaa.yang` and `ietf-netconf-acm.yang`. This way we can choose to export and and expose an entirely different AAA model.

Yet another very easy way out, is to define a set of static AAA rules whereby a set of fixed users and fixed groups have fixed access to our configuration data. Possibly the only field we wish to manipulate is the password field.

Chapter 15. The NETCONF Server

15.1. Introduction

This chapter describes the north bound NETCONF implementation in ConfD. As of this writing, the server supports the following specifications:

- RFC 4741 - NETCONF Configuration Protocol
- RFC 4742 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 5277 - NETCONF Event Notifications
- RFC 5717 - Partial Lock Remote Procedure Call (RPC) for NETCONF
- RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
- RFC 6021 - Common YANG Data Types
- RFC 6022 - YANG Module for NETCONF Monitoring
- RFC 6241 - Network Configuration Protocol (NETCONF)
- RFC 6242 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 6243 - With-defaults capability for NETCONF
- RFC 6470 - NETCONF Base Notifications
- RFC 6536 - NETCONF Access Control Model
- RFC 6991 - Common YANG Data Types

The ConfD NETCONF north bound API can be used by arbitrary NETCONF clients. A simple Python based NETCONF client called `netconf-console` is shipped as source code in the distribution. See Section 15.8, “Using netconf-console” for details. Other NETCONF clients will work too, as long as they adhere to the NETCONF protocol. If you need a Java client, the open source client JNC can be used.

When integrating NCS into larger OSS/NMS environments, the NETCONF API is a good choice of integration point.

15.2. Capabilities

The NETCONF server in ConfD supports all capabilities in both NETCONF 1.0 (RFC 4741) and NETCONF 1.1 (RFC 6241).

`:writable-running`
: This capability is enabled by default. If the candidate is used, this capability should be disabled in confd.conf(5). Additionally, `/confdConfig/datastores/running/access` should be set to *writable-through-candidate*.

`:candidate`
: The NETCONF server uses the candidate provided by the ConfD backplane. This can either be implemented in an external database, or using the built-in candidate support.

This capability is enabled by default. If the candidate is not used, this capability should be disabled in confd.conf(5).

:confirmed-commit
If the *running* data store is implemented as an external database, it has to support the *checkpoint* functions (see Chapter 7, *The external database API*). If it doesn't support checkpoints, this capability must be disabled. The built-in CDB database supports checkpoints, and can thus be used with this capability.

This capability is enabled by default. If the candidate is not used, this capability should be disabled in confd.conf(5).

ConfD supports both version 1.0 and 1.1 of this capability.

:rollback-on-error
This capability allows the client to set the `<error-option>` parameter to `rollback-on-error`. The other permitted values are `stop-on-error` (default) and `continue-on-error`. Note that the meaning of the word "error" in this context is not defined in the specification. Instead, the meaning of this word must be defined by the data model. Also note that if `stop-on-error` or `continue-on-error` is triggered by the server, it means that some parts of the edit operation succeeded, and some parts didn't. The error `partial-operation` must be returned in this case. If some other error occurs (i.e. an error not covered by the meaning of "error" above), the server generates an appropriate error message, and the data store is unaffected by the operation.

The ConfD server never allows partial configuration changes, since it might result in inconsistent configurations, and recovery from such a state can be very difficult for a client. This means that regardless of the value of the `<error-option>` parameter, ConfD will always behave as if it had the value `rollback-on-error`. So in ConfD, the meaning of the word "error" in `stop-on-error` and `continue-on-error`, is something which never can happen.

This capability is enabled by default. It can be disabled in confd.conf(5), but it doesn't affect the server behavior, other than the capability is not advertised.

:validate
This capability is enabled by default. It can be disabled in confd.conf(5). The only reason for disabling this capability would be if CDB is not used, and validation constraints are not specified in the YANG data models, and the underlying database does not support any form of validation.

ConfD supports both version 1.0 and 1.1 of this capability.

:startup
This capability is disabled by default. Enable this if `/confdConfig/datastores/startup` is enabled.

:url
The URL schemes supported are *file*, *ftp*, and *sftp* (SSH File Transfer Protocol).

There is no standard URL syntax for the *sftp* scheme, but ConfD supports the syntax used by `curl`:

```
sftp://<user>:<password>@<host>/<path>
```

Note that user name and password must be given for *sftp* URLs.

This capability is disabled by default, but can be enabled in confd.conf(5).

:xpath
This capability is enabled by default, but can be disabled in confd.conf(5).

The NETCONF server supports XPath according to the W3C XPath 1.0 specification (http://www.w3.org/TR/xpath), except for the list given below. There are several reasons for not supporting conventional XPath or for diverging from XPath, including the following:

1. The operation is performed on an XML database, not an XML document.

2. The implementation context does not support the operation.

3. Immaturity of IETF specifications. This refers to the result returned for some queries.

An XPath expression evaluation may terminate without matches or with an error (returned as a NETCONF error). Upon one or more successful matches, the XPath output is returned as an XML tree summarizing the matched database information, similarly to a conventional NETCONF subtree filter.

The following XPath features are not available:

- Variables are not supported, since the evaluation context binds no variables.

- Some location step axes are not supported: preceding, following, preceding-sibling, following-sibling.

- Some node tests are not supported: comment(), processing-instruction(). Note that these node types are not stored in the database.

- The XPath root node is not available. Instead, evaluation begins from each exported namespace. This primarily affects the parent and ancestor axes.

- XPath built-ins:

  `id()` the database does not store unique IDs

The following list of optional standard capabilities are also supported:

`:notification`
ConfD implements the `urn:ietf:params:netconf:capability:notification:1.0` capability, including support for the optional replay feature.

This capability is disabled by default, but can be enabled in confd.conf(5).

See Section 15.7, “Notification Capability” for details.

`:interleave`
ConfD implements the `urn:ietf:params:netconf:capability:interleave:1.0` capability, which allows the client to get send RPCs while a notification subscription is active.

This capability is disabled by default, but can be enabled in confd.conf(5).

`:partial-lock`
ConfD implements the `urn:ietf:params:netconf:capability:partial-lock:1.0` capability, which allows the client to lock parts of the running data store.

This capability is disabled by default, but can be enabled in confd.conf(5).

`:with-defaults`
ConfD implements the `urn:ietf:params:netconf:capability:with-defaults:1.0` capability, which is used by the server to inform the client how default values are

handled by the server, and by the client to control whether defaults values should be generated to replies or not.

This capability is enabled by default, but can be disabled in confd.conf(5).

In addition to the standard capabilities ConfD also includes the following optional, non-standard capabilities. They must be explicitly enabled in `confd.conf` (see confd.conf(5)) to be used.

`actions`
See Section 15.9, “Actions Capability” for details.

This capability should be enabled if actions are defined in the data model.

`transactions`
See Section 15.10, “Transactions Capability” for details.

This capability should be defined if ConfD runs as a subagent.

`proxy forwarding`
See Section 15.11, “Proxy Forwarding Capability” for details. This capability should be defined if ConfD runs as a proxy host.

`inactive`
See Section 15.12, “Inactive Capability” for details.

This capability should be defined if ConfD is configured to use attributes (see `/confdConfig/enableAttributes` in confd.conf(5)).

The server reports each data model namespace it has loaded as separate capabilities, according to the YANG specification.

The user can configure the server to make it report additional capability URIs.

15.3. NETCONF Transport Protocols

The NETCONF server natively supports the mandatory SSH transport, i.e., SSH is supported without the need for an external SSH daemon (such as sshd). It also support integration with OpenSSH.

15.3.1. Using OpenSSH

ConfD is delivered with a program **netconf-subsys** which is an OpenSSH *subsystem* program. It is invoked by the OpenSSH daemon after successful authentication. It functions as a relay between the ssh daemon and ConfD; it reads data from the ssh daemon from standard input, and writes the data to ConfD over a loopback socket, and vice versa. This program is delivered as source code in `$CONFD_DIR/src/confd/netconf/netconf-subsys.c` . It can be modified to fit the needs of the application. For example, it could be modified to read the group names for a user from an external LDAP server.

When using OpenSSH, the users are authenticated by OpenSSH, i.e. the user names are not stored in ConfD. To use OpenSSH, compile the **netconf-subsys** program, and put the executable in e.g. `/usr/local/bin`. Then add the following line to the ssh daemon's config file, `sshd_config`:

```
Subsystem     netconf   /usr/local/bin/netconf-subsys
```

Make sure ConfD is configured to listen to TCP traffic on localhost, port 2023, and disable SSH in `confd.conf` (see confd.conf(5)). (Re)start sshd and ConfD.

By default the **netconf-subsys** program sends the names of the UNIX groups the authenticated user belongs to. To test this, make sure that ConfD is configured to give access to the group(s) the user belongs to. Easiest for test is to give access to all groups.

15.3.2. Internal TCP Transport

The server can also be configured to accept plain TCP traffic. This can be useful during development, for debugging purposes, but it can also be used to plug in any other transport protocol. The way this works is that some other daemon terminates the transport and authenticates the user. Then it connects to the NETCONF server over TCP (preferably over the loopback interface for security reasons) and relays the XML traffic to NETCONF.

In this case, the transport daemon will have to authenticate the user, and then tell the NETCONF server about it. This should be done as a header sent over the TCP socket before any other bytes are sent. The header looks like this:

```
[username;source;proto;uid;gid;subgids;homedir;group-list;]\n
```

`username` is the name of the authenticated user. *`source`* is the textual representation of the ipv4 or ipv6 address and port which the user connected from, with address and port separated by '/' (e.g. "10.0.0.1/1234"). *`proto`* is the name of the transport protocol the client used (e.g. "beep" or "ssh"). *`uid`*, *`gid`*, *`supgids`* and *`homedir`* are the UNIX user id, group id, supplementary group ids and home directory for this user. These four parameters are only used if the user invokes a NETCONF RPC which is implemented with an external program (see Section 15.5, “Extending the NETCONF Server”). *`group-list`* is a comma-separated list of group names for the user. This list should only be sent if the transport has the capability to determine which groups a user belongs to. If not, an empty list should be sent. In this case, the normal AAA mechanisms are used to determine group membership.

All NETCONF RPCs sent over this socket must use the framing protocol used by NETCONF over SSH.

The TCP socket is also used if we want to use a standard SSH daemon such as sshd instead of the built-in SSH implementation. Then we would configure sshd to invoke a special program for the "netconf" subsystem. This special program would connect to the TCP socket as described above. See more below.

15.4. Configuration of the NETCONF Server

ConfD itself is configured through a configuration file called `confd.conf`. In that file the following items are related to the NETCONF server. For a complete description of these parameters, please see the confd.conf(5) man page.

`/confdConfig/logs/netconfLog`	This log can be enabled in order to troubleshoot the netconf sessions.
`/confdConfig/logs/netconfTraceLog`	When this log is enabled, all NETCONF traffic to and from ConfD is stored in a file. This can be useful in order to understand and troubleshoot the NETCONF protocol interactions.
`/confdConfig/aaa/sshServerKeyDir`	This is where the built-in SSH server reads its ssh keys.
`/confdConfig/aaa/pam/service`	This is the name of the PAM service to be used by the built-in SSH server. Used only if PAM is enabled (which means an SSH user can log in with username and password).

`/confdConfig/netconf/enabled`	When set to "true", the NETCONF server is started.
`/confdConfig/netconf/transport/ssh`	Settings for the built-in SSH server, such as listen ip address and port.
`/confdConfig/netconf/transport/tcp`	Settings for the plain-text TCP transport, such as listen ip address and port.
`/confdConfig/netconf/capabilities`	Under this parameter, we can control which capabilities are reported by the server.
`/confdConfig/netconf/capabilities/capability`	This parameter can be given multiple times. It specifies a URI string which the NETCONF server will report as a capability in the hello message sent to the client.

15.4.1. Error Handling

When ConfD processes `<get>`, `<get-config>`, and `<copy-config>` requests, the resulting data set can be very large. To avoid buffering huge amounts of data, ConfD streams the reply to the client as it traverses the data tree and calls data provider functions to retrieve the data.

If a data provider fails to return the data it is supposed to return, ConfD can take one of two actions. Either it simply closes the NETCONF transport (default), or it can reply with an *inline rpc error* and continue to process the next data element. This behavior can be controlled with the `/confdConfig/netconf/rpcErrors` configuration parameter (see confd.conf(5)). .

An inline error is always generated as a child element to the parent of the faulty element. For example, if an error occurs when retrieving the leaf element "mac-address" of an "interface" the error might be:

```
<interface>
  <name>atm1</name>
  <rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <error-type>application</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-message xml:lang="en">Failed to talk to hardware</error-message>
    <error-info>
      <bad-element>mac-address</bad-element>
    </error-info>
  </rpc-error>
  ...
</interface>
```

If a `get_next` call fails in the processing of a list, a reply might look like this:

```
<interface>
  <!-- successfully retrieved list entry -->
  <name>eth0</name>
  <mtu>1500</mtu>
  <!-- more leafs here -->
</interface>
<rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <error-type>application</error-type>
  <error-tag>operation-failed</error-tag>
  <error-severity>error</error-severity>
  <error-message xml:lang="en">Failed to talk to hardware</error-message>
```

```
  <error-info>
    <bad-element>interface</bad-element>
  </error-info>
</rpc-error>
```

15.5. Extending the NETCONF Server

NETCONF is an extensible protocol in the sense that new RPC operations can be defined separately from the standard. The NETCONF server in ConfD supports this through a simple API. New operations are typically identified with a new capability. When a new capability is implemented in this way, the name of the capability should be added to the list of capabilities that the NETCONF server sends in its initial `<hello>` message. This list is defined in `confd.conf` (see confd.conf(5)).

New RPCs are defined in YANG modules.

An RPC can be implemented in three different ways:

- As an executable program which is started by ConfD for each new RPC. The XML is passed as-is from ConfD to the program, and the resulting XML is generated by the program.
- As an executable program which is started by ConfD for each new RPC. The XML is parsed by ConfD, and passed (in a certain format) on the command line to the program. ConfD generates an XML reply based on the result from the program.
- As a C callback function. The application registers the callback with ConfD, and ConfD invokes the callback function when the RPC operation is received. ConfD parses the XML and passes it in a C data structure to the callback. ConfD generates an XML reply from the return value from the callback.

15.5.1. RPC as an Executable, ConfD does not Translate the XML

In this case, the RPC is implemented as an ordinary executable program, which communicate with ConfD over stdin/stdout. When ConfD invokes the program, it will pass the entire XML operation on stdin. The program is responsible for parsing the operation data and placing its reply on stdout, and then terminate with exit status zero. ConfD wraps this reply in a `<rpc-reply>` element. Note that ConfD does not interpret the reply XML sent by the program; it merely sends the data as-is to the NETCONF client. Thus, it is the responsibility of the program to produce a valid NETCONF XML reply. Note that a rpc reply MUST contain one of `<ok/>`, `<data>` or `<rpc-error>`.

A program can also be run in *batch mode*, which can be used to send asynchronous data to the client. In this case, the program does not exit after having replied to the original RPC. Instead it signals that the reply has been sent by sending a NUL byte to ConfD. ConfD will enter its main loop and listen for new requests from the client and data from the external programs. When data is received from one source, this source is handled, while the others are (potentially) blocked. The asynchronous data sent by the external program must be a complete self-contained XML chunk, followed by a single NUL byte. The program can exit at any time, the session towards the client is not terminated just because the program exits.

The maximum number of concurrently running batch processes can be set in `confd.conf` (see confd.conf(5)) using the parameter `/confdConfig/netconf/maxBatchProcesses`. The default is no limit.

Here's an example of an rpc operation defined in this way:

```
module math-rpc {
  namespace "http://example.com/math/1.0";
  prefix math;

  import tailf-common {
    prefix tailf;
  }

  rpc math {
    tailf:exec "/usr/bin/math" {
      tailf:raw-xml;
    }
  }
}
```

All these examples are available under `netconf_extensions/simple_rpc` in the examples distribution.

Now suppose that the following rpc is received by the NETCONF server:

Example 15.1. Example math rpc

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <math xmlns="http://example.com/math/1.0">
    <add>
      <operand>2</operand>
      <operand>3</operand>
    </add>
  </math>
</rpc>
```

ConfD will invoke `/usr/local/bin/math` and pass:

```
<math xmlns="http://example.com/math/1.0">
  <add>
    <operand>2</operand>
    <operand>3</operand>
  </add>
</math>
```

on stdin. The program will print the rpc-reply to stdout, and ConfD relays this data to the client.

15.5.2. RPC as an Executable, ConfD Translates the XML

In this case, the RPC is implemented as an ordinary executable program, with all XML parameters converted by ConfD into command-line arguments to the program. If the program terminates normally without producing any output on stdout, ConfD replies with an `<ok/>` rpc-reply. If the program terminates normally and also generates data on stdout, ConfD interprets this data and passes it with `<data>` tags. If the program terminates abnormally without producing any data, a generic `operation-failed` error is returned. Finally, if the program terminates abnormally and also generates data on stdout, ConfD interprets this data as an rpc-error, and sends the resulting XML to the client.

Here's an example of the same rpc operation as above defined this way:

```
module math-rpc {
  namespace "http://example.com/math/1.0";
  prefix math;
```

```
  import tailf-common {
    prefix tailf;
  }

  rpc math {
    tailf:exec "/usr/bin/math";
    input {
      choice op {
        container add {
          leaf-list operand {
            type int32;
            min-elements 2;
            max-elements 2;
          }
        }
        container sub {
          leaf-list operand {
            type int32;
            min-elements 2;
            max-elements 2;
          }
        }
      }
    }
    output {
      leaf result {
        type int32;
      }
    }
  }
}
```

Now suppose that the same RPC request as in Code listing 2 above is received. ConfD parses the XML and invokes the command as:

```
/usr/local/bin/math add __BEGIN operand 2 operand 3 add __END
```

In general, the XML is flattened, and each XML element generates two strings on the command line. If a container is received, the strings "elem-name" "__BEGIN" is generated. When the corresponding close element is received, "elem-name" "__END" is generated. An element with a value will generate "elem-name" "value". An empty element with no subelements will generate "elem-name" "__LEAF".

Next, the math program replies by printing on stdout:

```
result 5
```

The same translation rules applies to the result, and ConfD thus sends the following reply to the client:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <result xmlns="http://example.com/math/1.0">5</result>
  </data>
</rpc-reply>
```

15.5.3. RPC as a Callback Function

In this case, the RPC is implemented as a callback function in C, with all XML parameters converted by ConfD into a C data structure.

Here's an example of the same rpc operation as above defined this way:

```
module math-rpc {
  namespace "http://example.com/math/1.0";
  prefix math;

  import tailf-common {
    prefix tailf;
  }

  rpc math {
    tailf:actionpoint "math";
    input {
      choice op {
        container add {
          leaf-list operand {
            type int32;
            min-elements 2;
            max-elements 2;
          }
        }
        container sub {
          leaf-list operand {
            type int32;
            min-elements 2;
            max-elements 2;
          }
        }
      }
    }
    output {
      leaf result {
        type int32;
      }
    }
  }
}
```

The code that implements this looks like this:

```
static void register_math(struct confd_daemon_ctx *dctx)
{
    struct confd_action_cbs acb;

    memset(&acb, 0, sizeof(acb));
    strcpy(acb.actionpoint, "math");
    acb.init = init_action;  /* this function is not shown here */
    acb.action = do_math;
    if (confd_register_action_cbs(dctx, &acb) != CONFD_OK)
        confd_fatal("Couldn't register action callbacks\n");
    if (confd_register_done(dctx) != CONFD_OK)
        confd_fatal("Failed to complete registration \n");
}

static int do_math(struct confd_user_info *uinfo,
                   struct xml_tag *name,
                   confd_hkeypath_t *kp,
                   confd_tag_value_t *params,
```

```
                  int nparams)
{
    confd_tag_value_t reply[1];
    int op1, op2, result;

    /*
      we know that we get exactly 4 parameters;
         add | del BEGIN
         operand 1
         operand 2
         add | del END
    */

    op1 = CONFD_GET_INT32(CONFD_GET_TAG_VALUE(&params[1]));
    op2 = CONFD_GET_INT32(CONFD_GET_TAG_VALUE(&params[2]));
    switch (CONFD_GET_TAG_TAG(&params[0])) {
    case math_add:
        result = op1 + op2;
        break;
    case math_del:
        result = op1 - op2;
        break;
    }

    CONFD_SET_TAG_INT32(&reply[0], math_result, result);
    confd_action_reply_values(uinfo, reply, 1);

    return CONFD_OK;
}
```

15.6. Monitoring of the NETCONF Server

RFC 6022 - YANG Module for NETCONF Monitoring defines a YANG module, `ietf-net-conf-monitoring`, for monitoring of the NETCONF server. It contains statistics objects such as number of RPCs received, status objects such as user sessions, and an operation to retrieve data models from the NETCONF server.

In order to use this data model with ConfD, the fxs file (`ietf-netconf-monitoring.fxs`) must be present in ConfD's loadPath. This fxs file is present in a development installation of ConfD.

This data model defines a new RPC operation, `get-schema`, which is used to retrieve YANG modules from the NETCONF server. ConfD will report the YANG modules for all fxs files that are reported as capabilities, and for which the corresponding YANG or YIN file is found in the loadPath. Thus, if this module is used, all fxs files, YANG, and YIN files must be copied into the loadPath. The YANG and YIN files can be stored as is or compressed with gzip. The filename extension MUST be ".yang", ".yin", ".yang.gz", or ".yin.gz".

Also available is a Tail-f specific data model, `tailf-netconf-monitoring`, which augments `ietf-netconf-monitoring` with additional data about files available for usage with the `<copy-config>` command with a *file* `<url>` source or target.

`/confdConfig/netconf/capabilities/url/enabled` and `/confdConfig/net-conf/capabilities/url/file/enabled` must both be set to true. If rollbacks are enabled, those files are listed as well, and they can be loaded using `<copy-config>`.

This data model also adds data about which notification streams are present in the system, and data about sessions that subscribe to the streams.

In order to use this data model with ConfD, the fxs file (`tailf-netconf-monitoring.fxs`) must be present in ConfD's loadPath. This fxs file is present in a development installation of ConfD.

These fxs files are available in the `$CONFD_DIR/etc/confd` directory, and the source for them are available in the `$CONFD_DIR/src/confd/yang` directory, in the distribution. The `Makefile` in the latter directory can be modified as necessary, for example to compile the fxs files with a *`--export`* parameter to **confdc**.

15.7. Notification Capability

This section describes how NETCONF notifications are implemented within ConfD, and how the applications generates these events.

Central to NETCONF notifications is the concept of a *stream*. The stream serves two purposes. It works like a high-level filtering mechanism for the client. For example, if the client subscribes to notifications on the `security` stream, it can expect to get security related notifications only. Second, each stream may have its own log mechanism. For example by keeping all debug notifications in a `debug` stream, they can be logged separately from the `security` stream.

15.7.1. Notification Streams

ConfD has built-in support for the well-known stream `NETCONF`, defined in RFC 5277. ConfD supports the notifications defined in RFC 6470 - NETCONF Base Notifications on this stream. If the application needs to send any additional notifications on this stream, it can do so.

It is up to the application to define which additional streams it supports. In ConfD, this is done in `confd.conf` (see confd.conf(5)). Each stream must be listed, and whether it supports replay or not. An example which defines two streams, `security` and `debug`:

```
<notifications>
  <eventStreams>
    <stream>
      <name>security</name>
      <description>Security related notifications</description>
      <replaySupport>true</replaySupport>
      <builtinReplayStore>
        <enabled>true</enabled>
        <dir>/var/log</dir>
        <maxSize>S10M</maxSize>
        <maxFiles>50</maxFiles>
      </builtinReplayStore>
    </stream>
    <stream>
      <name>debug</name>
      <description>Debug notifications</description>
      <replaySupport>true</replaySupport>
    </stream>
  </eventStreams>
</notifications>
```

The well-known stream `NETCONF` does not have to be listed, but if it isn't listed, it will not support replay.

15.7.2. Automatic Replay

ConfD has builtin support for logging of notifications, i.e., if replay support has been enabled for a stream, ConfD automatically stores all notifications on disk ready to be replayed should a NETCONF client ask

for logged notifications. In the `confd.conf` fragment above the security stream has been setup to use the builtin notification log/replay store. The replay store uses a set of wrapping log files on disk (of a certain number and size) to store the security stream notifications.

The reason for using a wrap log is to improve replay performance whenever a NETCONF client asks for notifications in a certain time range. Any problems with log files not being properly closed due to hard power failures etc. is also kept to a minimum, i.e., automatically taken care of by ConfD.

As an alternative to the builtin notification replay store the application can roll its own. This is described in the next sub-section.

15.7.3. Implementing Custom Replay

If a stream supports replay, the logging and replay functionality can alternatively be implemented by the application. In order to do this, the application must register a set of callback functions with ConfD using the function `confd_register_notification_stream()`. The callbacks are `get_log_start_time()` and `replay()`. The first one is called by ConfD in order to find the earliest event time available in the log. The second one is invoked whenever a NETCONF client asks for a replay subscription. For full details on the notification API, please see the confd_lib_dp(3) manual page.

The following example is available in full source code form in the examples directory. A single stream `interface` is used, and it supports replay.

```
/* The notification context (filled in by ConfD) for the live feed */
static struct confd_notification_ctx *live_ctx;


          struct confd_notification_stream_cbs ncb;


          memset(&ncb, 0, sizeof(ncb));
    ncb.fd = workersock;
    ncb.get_log_times = log_times;
    ncb.replay = start_replay;
    strcpy(ncb.streamname, "interface");
    ncb.cb_opaque = NULL;
    if (confd_register_notification_stream(dctx, &ncb, &live_ctx) != CONFD_OK) {
        confd_fatal("Couldn't register stream %s\n", ncb.streamname);
    }
    if (confd_register_done(dctx) != CONFD_OK) {
        confd_fatal("Failed to complete registration\n");
    }
```

In this simple example, we keep the replay log in memory, in an array:

```
struct notif {
    struct confd_datetime eventTime;
    confd_tag_value_t *vals;
    int nvals;
};

/* Our replay buffer is kept in memory in this example.  It's a circular
 * buffer of struct notif.
 */
#define MAX_BUFFERED_NOTIFS 4
static struct notif replay_buffer[MAX_BUFFERED_NOTIFS];
static unsigned int first_replay_idx = 0;
```

```
static unsigned int next_replay_idx = 0;

static struct confd_datetime replay_creation;
static int replay_has_aged_out = 0;
static struct confd_datetime replay_aged_time;
```

The `get_log_start_time()` callback simply returns the time of the first notification in the log:

```
static int log_times(struct confd_notification_ctx *nctx)
{
    struct confd_datetime *aged;

    if (replay_has_aged_out)
        aged = &replay_aged_time;
    else
        aged = NULL;

    return confd_notification_reply_log_times(nctx, &replay_creation, aged);
}
```

When a client asks for a replay subscription, ConfD invokes the callback `replay`. The actual replay notifications must not be sent from the callback. In this example, the callback allocates a replay structure, and marks it as being active. The main loop will check for any active replays, and do the sending there.

```
#define MAX_REPLAYS 10
struct replay {
    int active;
    int started;
    unsigned int idx;
    struct confd_notification_ctx *ctx;
    struct confd_datetime start;
    struct confd_datetime stop;
    int has_stop;
};
/* Keep tracks of active replays */
static struct replay replay[MAX_REPLAYS];


      static int start_replay(struct confd_notification_ctx *nctx,
                        struct confd_datetime *start,
                        struct confd_datetime *stop)
{
    int rnum;

    for (rnum = 0; rnum < MAX_REPLAYS; rnum++) {
        if (!replay[rnum].active) {
            replay[rnum].active = 1;
            replay[rnum].started = 0;
            replay[rnum].idx = first_replay_idx;
            replay[rnum].ctx = nctx;
            replay[rnum].start = *start;
            if (stop) {
                replay[rnum].has_stop = 1;
                replay[rnum].stop = *stop;
            } else
                replay[rnum].has_stop = 0; /* stop when caught up to live */
            return CONFD_OK;
        }
    }
```

```
    confd_notification_seterr(nctx, "Max no. of replay requests reached");
    return CONFD_ERR;
}
```

15.7.4. Sending Notifications from an Application

Before an application can send a notification, the notification must be defined in a YANG module. In this example, a notification `link-down` is defined. The notification has a single parameter *`if-index`*:

```
  notification linkDown {
    leaf ifIndex {
      type leafref {
        path "/interfaces/interface/ifIndex";
      }
      mandatory true;
    }
  }
```

When the application sends an application, it uses the function `confd_notification_send()`.

```
static void send_notifdown(int index)
{
    confd_tag_value_t vals[3];
    int i = 0;

    CONFD_SET_TAG_XMLBEGIN(&vals[i], notif_linkDown,     notif__ns);  i++;
    CONFD_SET_TAG_UINT32(&vals[i],   notif_ifIndex,      index);      i++;
    CONFD_SET_TAG_XMLEND(&vals[i],   notif_linkDown,     notif__ns);  i++;
    send_notification(vals, i);
}


      static void send_notification(confd_tag_value_t *vals, int nvals)
{
    int sz;
    struct confd_datetime now;
    struct notif *notif;

    getdatetime(&now);
    notif = &replay_buffer[next_replay_idx];
    if (notif->vals) {
        /* we're aging out this notification */
        replay_has_aged_out = 1;
        replay_aged_time = notif->eventTime;
        first_replay_idx = (first_replay_idx + 1) % MAX_BUFFERED_NOTIFS;
        free(notif->vals);
    }
    notif->eventTime = now;
    sz = nvals * sizeof(confd_tag_value_t);
    notif->vals = malloc(sz);
    memcpy(notif->vals, vals, sz);
    notif->nvals = nvals;
    next_replay_idx = (next_replay_idx + 1) % MAX_BUFFERED_NOTIFS;
    OK(confd_notification_send(live_ctx,
                               &notif->eventTime,
                               notif->vals,
                               notif->nvals));
}
```

15.8. Using netconf-console

The `netconf-console` program is a simple NETCONF client. It is delivered as Python source code and can be used as-is or modified.

When ConfD has been started, we can use `netconf-console` to query the configuration of the NETCONF Access Control groups:

```
$ netconf-console --get-config -x /nacm/groups
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm">
      <groups>
        <group>
          <name>admin</name>
          <user-name>admin</user-name>
          <user-name>private</user-name>
        </group>
        <group>
          <name>oper</name>
          <user-name>oper</user-name>
          <user-name>public</user-name>
        </group>
      </groups>
    </nacm>
  </data>
</rpc-reply>
```

With the `-x` flag an XPath expression can be specified, in order to retrieve only data matching that expression. This is a very convenient way to extract portions of the configuration from the shell or from shell scripts.

15.9. Actions Capability

15.9.1. Overview

This capability introduces one new rpc method which is used to invoke actions (methods) defined in the data model. When an action is invoked, the instance on which the action is invoked is explicitly identified by an hierarchy of configuration or state data.

Here's a simple example which resets an interface.

```
  <rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <action xmlns="http://tail-f.com/ns/netconf/actions/1.0">
      <data>
        <interfaces xmlns="http://example.com/interfaces/1.0">
          <interface>
            <name>eth0</name>
            <reset/>
          </interface>
        </interfaces>
      </data>
```

```
  </action>
</rpc>

<rpc-reply message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.9.2. Motivation

The alternative is to use a specialized rpc method for each action. There are a couple of drawbacks with that:

- The name of the action has to be unique within the namespace. With the generic action method, the name of the action is scoped by the element where the action is defined. For example, without a generic action, ther might be two rpcs, 'reset-interface' and 'reset-server'. With the generic action, there are two 'reset' actions, scoped by 'interface' and 'server'.
- Care must be taken to ensure that returned XML is unique within the namespace. Suppose the two methods 'reset-interface' and 'reset-server' returns a 'status', but of different type. The element must be called something like 'reset-interface-status' and 'reset-server-status'.
- With the generic action, it is easier to introduce intermediate NETCONF peers such as a master agent in a master-sub agent deployment. For example, suppose there are two subagents, one which handles interface 'eth0' and one which handles 'atm0'. When the hierarchy is excplicit in the request, the master agent can dispatch to the correct subagent without any knowledge about the action parameters. On the other hand, if the master agent gets a rpc 'reset-interface', it will have to parse the parameters to figure out which subagent to send the request to.

15.9.3. Dependencies

None.

15.9.4. Capability Identifier

The actions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/actions/1.0
```

15.9.5. New Operation: <action>

Description

The <action> operation identifies the data instance where the action is invoked, the action name, and its parameters. If the action returns any result, it is scoped in the instance hierarchy in the reply.

Parameters

`data:` A hierarchy of configuration or state data as defined by one of the device's data models. The first part of the hierarchy defines which instance the action is invoked upon. Then comes the action name, and any parameters it might need.

One action only can be executed within one rpc. If more than one actions are present in the rpc, an error MUST be returned with an <error-tag> set to "bad-element".

Positive Response

An action that does not return any result value, replies with the standard <ok/>. If a result value is returned, it is encapsulated in the standard <data> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

Example

Suppose we want to start a self-test on interface "eth0", and the test returns the run time (in seconds) of the test and test status. In pseudo code

```
myif = find_if("eth0")
(time, status) = myif.self_test(IMMEDIATELY)
```

Using the action RPC over NETCONF:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <action xmlns="http://tail-f.com/ns/netconf/actions/1.0">
    <data>
      <interfaces xmlns="http://example.com/interfaces/1.0">
        <interface>
          <name>eth0</name>
          <self-test>
            <when>immediately</when>
          </self-test>
        </interface>
      </interfaces>
    </data>
  </action>
</rpc>

<rpc-reply message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <interfaces xmlns="http://example.com/interfaces/1.0">
      <interface>
        <name>eth0</name>
          <self-test>
            <run-time>29</run-time>
            <status>ok</status>
          </self-test>
        </interface>
      </interfaces>
    </data>
  </action>
</rpc-reply>
```

15.9.6. Modifications to Existing Operations

None.

15.9.7. XML Schema

This XML Schema defines the new action rpc.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/actions/1.0"
    xmlns="http://tail-f.com/ns/netconf/actions/1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
    xml:lang="en">

  <!--  <action> operation -->
  <xs:complexType name="ActionType">
    <xs:complexContent>
      <xs:extension base="netconf:rpcOperationType">
        <xs:sequence>
          <xs:element name="data" type="netconf:dataInlineType" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="action" type="ActionType"
              substitutionGroup="netconf:rpcOperation"/>

</xs:schema>
```

15.10. Transactions Capability

15.10.1. Overview

This capability introduces four new rpc methods which are used to control a two-phase commit transaction on the NETCONF server. The normal <edit-config> operation is used to write data in the transaction, but the modifications are not applied until an explicit <commit-transaction> is sent.

This capability is formally defined in the YANG module "tailf-netconf-transactions".

A typical sequence of operations looks like this:

```
              C                           S
              |                           |
              |  capability exchange      |
              |-------------------------->|
              |<--------------------------|
              |                           |
              |   <start-transaction>     |
              |-------------------------->|
              |<--------------------------|
              |         <ok/>             |
              |                           |
              |     <edit-config>         |
              |-------------------------->|
              |<--------------------------|
              |         <ok/>             |
              |                           |
              |  <prepare-transaction>    |
              |-------------------------->|
              |<--------------------------|
              |         <ok/>             |
              |                           |
```

```
               |   <commit-transaction>    |
               |-------------------------->|
               |<--------------------------|
               |         <ok/>             |
               |                           |
```

15.10.2. Dependencies

None.

15.10.3. Capability Identifier

The transactions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/transactions/1.0
```

15.10.4. New Operation: <start-transaction>

Description

Starts a transaction towards a configuration datastore. There can be a single ongoing transaction per session at any time.

When a transaction has been started, the client can send any NETCONF operation, but any <edit-config> or <copy-config> operation sent from the client MUST specify the same <target> as the <start-transaction>, and any <get-config> MUST specify the same <source> as <start-transaction>.

If the server receives an <edit-config> or <copy-config> with another <target>, or a <get-config> with another <source>, an error MUST be returned with an <error-tag> set to "invalid-value".

The modifications sent in the <edit-config> operations are not immediately applied to the configuration datastore. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

The client sends a <prepare-transaction> when all modifications have been sent.

Parameters

target: Name of the configuration datastore towards which the transaction is started.

with-inactive: If this parameter is given, the transaction will handle the "inactive" and "active" attributes. If given, it MUST also be given in the <edit-config> and <get-config> invocations in the transaction.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is an ongoing transaction for this session already, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <start-transaction xmlns="http://tail-f.com/ns/netconf/transactions/1.0">
    <target>
     <running/>
    </target>
  </start-transaction>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.10.5. New Operation: <prepare-transaction>

Description

Prepares the transaction state for commit. The server may reject the prepare request for any reason, for example due to lack of resources or if the combined changes would result in an invalid configuration datastore.

After a successful <prepare-transaction>, the next transaction related rpc operation must be <commit-transaction> or <abort-transaction>. Note that an <edit-config> cannot be sent before the transaction is either committed or aborted.

Care must be taken by the server to make sure that if <prepare-transaction> succeeds then the <commit-transaction> SHOULD not fail, since this might result in an inconsistent distributed state. Thus, <prepare-transaction> should allocate any resources needed to make sure the <commit-transaction> will succeed.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <prepare-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>
```

```
<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.10.6. New Operation: <commit-transaction>

Description

Applies the changes made in the transaction to the configuration datatore. The transaction is closed after a <commit-transaction>.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has not been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.10.7. New Operation: <abort-transaction>

Description

Aborts the ongoing transaction, and all pending changes are discarded. <abort-transaction> can be given at any time during an ongoing transaction.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <abort-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.10.8. Modifications to Existing Operations

The <edit-config> operation is modified so that if it is received during an ongoing transaction, the modifications are not immediately applied to the configuration target. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

Note that it doesn't matter if the <test-option> is 'set' or 'test-then-set' in the <edit-config>, since nothing is actually set when the <edit-config> is received.

15.10.9. XML Schema

This XML Schema defines the new transaction rpcs.

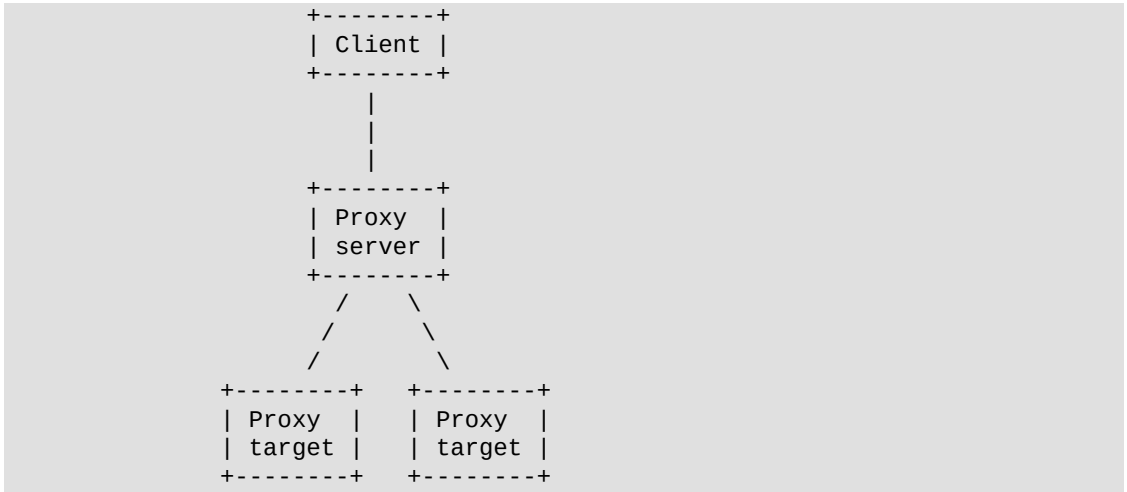
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/transactions/1.0"
    xmlns="http://tail-f.com/ns/netconf/transactions/1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
    xml:lang="en">

  <!-- Type for <target> element -->
  <xs:complexType name="TargetType">
    <xs:choice>
      <xs:element name="running"/>
      <xs:element name="startup"/>
      <xs:element name="candidate"/>
    </xs:choice>
  </xs:complexType>

  <!--  <start-transaction> operation -->
  <xs:complexType name="StartTransactionType">
    <xs:complexContent>
      <xs:extension base="netconf:rpcOperationType">
        <xs:sequence>
```

```
          <xs:element name="target" type="TargetType"/>
          <xs:element name="with-inactive" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="start-transaction" type="StartTransactionType"
              substitutionGroup="netconf:rpcOperation"/>

  <xs:element name="prepare-transaction"
              substitutionGroup="netconf:rpcOperation"/>

  <xs:element name="commit-transaction"
              substitutionGroup="netconf:rpcOperation"/>

  <xs:element name="abort-transaction"
              substitutionGroup="netconf:rpcOperation"/>

</xs:schema>
```

15.11. Proxy Forwarding Capability

15.11.1. Overview

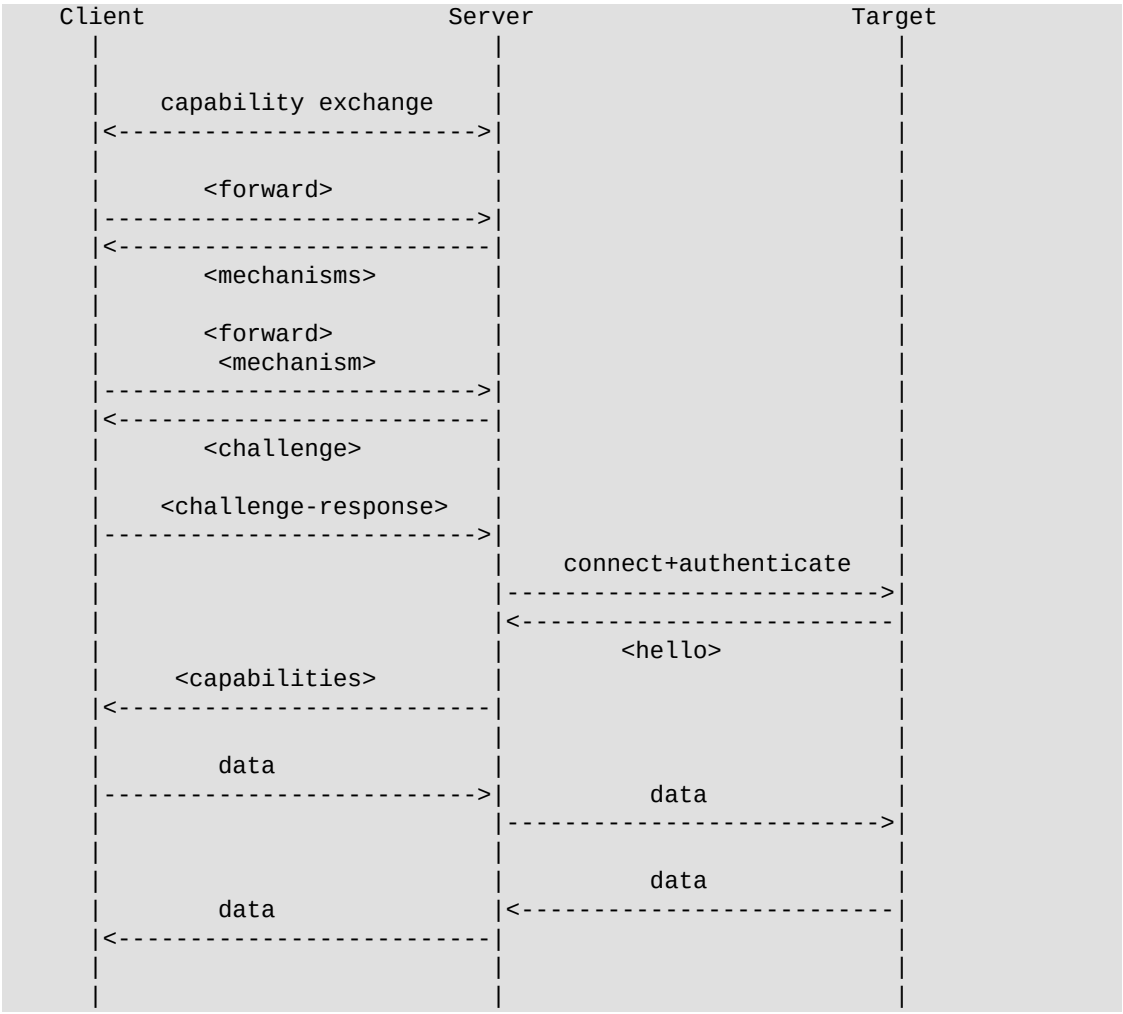
The Proxy Forwarding capability makes it possible to forward NETCONF requests to a target host through a proxy NETCONF server. It can be used in situations where a client does not have direct network access to a target host:



See RFC 2663 for a definition of a proxy. This RFC defines two terms "Application Level Gateway" (ALG) and "Proxy":

```
   ALGs are similar to Proxies, in that, both ALGs and proxies
   facilitate Application specific communication between clients
   and servers. Proxies use a special protocol to communicate with
   proxy clients and relay client data to servers and vice
   versa. Unlike Proxies, ALGs do not use a special protocol to
   communicate with application clients and do not require changes
   to application clients.
```

A client that wants to set up a NETCONF session to a Proxy target first connects to the Proxy server, which advertises the "forward" capability. The client issues a <forward> RPC, with a <target> parameter which specifies which Proxy target to connect to. The Proxy server sets up a NETCONF connection to the Proxy target, and after successful authentication, replies with the Proxy target's capability list to the client. From this point, the session is established, and any data received by the Proxy server from any side is sent as-is (without interpretation) to the other side.



Client Elements of Procedure

First, the client constructs a <forward> rpc:

1. If the client does not know with authentication mechanism is supported by the Proxy server for the target, or if it wants to do automatic login, it sends a <forward> request without the "auth" parameter, and waits for a reply.

2. If the client knows which mechanism to use, it sends a <forward> request with the "auth/mechanism" parameter set, and waits for a reply.

   The client MAY set the "auth/initial-response" parameter.

3. Then the client waits for a reply.

4. If the reply contains the "capabilities" parameter, the proxy connection is establihed.

5. If the reply contains the "challenge" parameter, the client sends a <challenge-response> RPC with the repsonse to the challenge, which it can get e.g. by prompting the user for credentials.

   If the mechanism is PLAIN, the challenge is always empty.

   After the <challenge-response> RPC is sent, the client continues from step (3).

6. If the reply contains the "sasl-failure" error, with the "failure" parameter set to "invalid-mechansim", the client continues from step (2).

7. If the reply contains the "sasl-failure" error, with the "failure" parameter set to "not-authorized", the client continues from step (1) or aborts.

8. Otherwise, the client interprets the error and aborts.

Server Elements of Procedure

The procedure when the <forward> RPC is received is as follows:

1. The server looks up the value of the "target" parameter in the "proxy" list in the running configuration. If the target is not found, an "invalid-value" error is returned.

2. If the "auth" parameter is not present, and the server is configured to perform auto login, it extracts the current user's credentials from the session, and continues from step (8).

3. If the "auth" parameter is not present, and the server is not configured to do auto login, it replies with an error "sasl-authentication-needed", with a list of supported mechanisms.

4. If the "auth" parameter is present, the server verifies that the "mechanism" provided is supported by the server.

   Currently, the supported mechanism is "PLAIN".

   If the mechanism is not supported, the server replies with a "sasl-failure" error with the "failure" parameter set to "invalid-mechanism".

5. If the mechanism is supported, and the "initial-response" parameter is present, the server decodes the response according to the mechanism.

   If the response could not be decoded, the server replies with an "sasl-failure" error with the "failure" parameter set to "incorrect-encoding".

   If the response could be decoded, the server continues from step (8).

6. If the mechanism is supported, and the "initial-response" parameter is not present, the server replies with a "challenge" parameter.

   For PLAIN, the challenge is empty.

   The server now remembers the target and mechanism, and waits to receive a <challenge-response> RPC.

7. When the <challenge-response> RPC is received, the server decodes the "response" parameter as in (5).

   If the response could be decoded, the server continues from step (8).

8. The server connects to the target with the given credentials.

If the connection fails due to communication problems, it replies with an "connection-failure" error.

If the server fails to authenticate with the given credentials, it replies with an "sasl-failure" error with the "failure" parameter set to "not-authorized".

If the connection succeeds, the server replies with the capabilities of the target, and enters the proxying mode.

In proxying mode, the server reads data from both the client and the target, and writes any data received to the other end, without interpreting the data. If any side of the connection is closed, the server closes the other side.

15.11.2. Dependencies

None.

15.11.3. Capability Identifier

The proxy forwarding capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/forward/1.0
```

15.11.4. New Operation: <forward>

Description

Starts a proxy forwarding connection to the given target, if all user credentials are given.

The server can be configured to automatically login to the target. In this case, the <forward> rpc does not contain any authentication parameters.

Parameters

target:	Name of the target host to connect to. The name refers to an entry in the "proxy" list on the running configuration.
auth/mechanism:	Name of an SASL authentication mechanism to use. Currently the "PLAIN" mechanism is supported.
auth/initial-response:	If allowed by the selected mechanism, an initial response can be given. This saves one round-trip. For the PLAIN mechanism, the response is a base64 encoded PLAIN "message" as defined in section 2 of RFC 4616. The optional "authzid" MUST NOT be present.

Positive Response

If the server was able to connect and authenticate to the target, it replies with the target's capability list, and the server then enters proxying mode.

If the server could not fully authenticate the client, it replies with a "challenge" element. The client should reply to the challenge with a <challenge-response> RPC.

Negative Response

If the server could not find the target, it replies with an "invalid-value" error.

If the client did not provide a mechanism, the server replies with a "sasl-authentication-needed" error, with a list of available mechanisms.

If the client provided an unsupported mechanism, the server replies with a "sasl-failure" error with the "failure" parameter set to"invalid-mechanism".

If the inital response could not be decoded, the server replies with an "sasl-failure" error with the "failure" parameter set to "incorrect-encoding".

If the server fails to connect to the target, it replies with an "connection-failure" error.

If the server fails to authenticate with the given credentials, it replies with an "sasl-failure" error with the "failure" parameter set to "not-authorized".

Example 1.

The proxy server is configured to do automatic login:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <data>
    <capabilities xmlns="http://tail-f.com/ns/netconf/forward/1.0">
      <capability>urn:ietf:params:netconf:base:1.0</capability>
      <capability>
        urn:ietf:params:netconf:capability:writable-running:1.0
      </capability
    </capabilities>
  </data>
</rpc-reply>

<!- client is now successfully connected to rne-141 -->
```

Example 2. Client needs to authenticate to the target:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
```

```
    <error-app-tag>sasl-mechanisms</error-app-tag>
    <error-info>
      <mechanisms xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <mechanism>PLAIN</mechanism>
      </mechansims>
    </error-info>
  </rpc-error>
</rpc>

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="2">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
    <auth>
      <mechanism>PLAIN</mechanism>
      <initial-response>AGFkbWluAHNlY3JldA==</initial-response>
    </auth>
  </forward>
</rpc>
```

The decoded initial response in the auth message is:

```
  <NUL>admin<NUL>secret

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="2">
  <data>
    <capabilities xmlns="http://tail-f.com/ns/netconf/forward/1.0">
      <capability>urn:ietf:params:netconf:base:1.0</capability>
    </capabilities>
  </data>
</rpc-reply>

<!- client is now successfully connected to rne-141 -->
```

Example 3. Client needs to authenticate to the proxy target, but fails:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-app-tag>sasl-authentication-needed</error-app-tag>
    <error-info>
      <mechanisms xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <mechanism>PLAIN</mechanism>
      </mechansims>
    </error-info>
  </rpc-error>
</rpc>
```

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="2">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
    <auth>
      <mechanism>PLAIN</mechanism>
      <initial-response>AGFkbWluAGFlY3JldA==</initial-response>
    </auth>
  </forward>
</rpc>
```

The decoded initial response in the auth message is:

```
  <NUL>admin<NUL>aecret   (bad passwd)

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="2">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-app-tag>sasl-failure</error-app-tag>
    <error-info>
      <failure xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <not-authorized/>
      </failure>
    </error-info>
  </rpc-error>
</rpc-reply>
```

15.11.5. New Operation: <challenge-response>

Description

Sent after receiving a challenge reply to the <forward> request. If it succeds, the server will enter proxying mode.

Parameters

response For the PLAIN mechanism, the response is a base64 encoded PLAIN "message" as defined in section 2 of RFC 4616. The optional "authzid" MUST NOT be present.

Positive Response

If the server was able to connect and authenticate to the target, it replies with the target's capability list, and the server then enters proxying mode.

If the server could not fully authenticate the client, it replies with a "challenge" element. The client should reply to the challenge with a <challenge-response> RPC.

Negative Response

If the response could not be decoded, the server replies with an "sasl-failure" error with the "failure" parameter set to "incorrect-encoding".

If the server fails to connect to the target, it replies with an "connection-failure" error.

If the server fails to authenticate with the given credentials, it replies with an "sasl-failure" error with the "failure" parameter set to "not-authorized".

Example

Client needs to authenticate to the target:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-app-tag>sasl-mechanisms</error-app-tag>
    <error-info>
      <mechanisms xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <mechanism>PLAIN</mechanism>
      </mechansims>
    </error-info>
  </rpc-error>
</rpc>

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="2">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>rne-141</target>
    <auth>
      <mechanism>PLAIN</mechanism>
    </auth>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="2">
  <data>
    <challenge xmlns="http://tail-f.com/ns/netconf/forward/1.0"/>
  </data>
</rpc-reply>

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="3">
  <challenge-response xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <reponse>AGFkbWluAHNlY3JldA==</response>
  </challenge-response>
</rpc>
```

The decoded response in the auth message is:

```
  <NUL>admin<NUL>secret
```

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="3">
  <data>
    <capabilities xmlns="http://tail-f.com/ns/netconf/forward/1.0">
      <capability>urn:ietf:params:netconf:base:1.0</capability>
    </capabilities>
  </data>
</rpc-reply>

<!- client is now successfully connected to rne-141 -->
```

15.11.6. Modifications to Existing Operations

None.

15.11.7. Interactions with Other Capabilities

None.

15.11.8. XSD Schema

This XML Schema defines the new forwarding rpcs.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/forward/1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tail-f.com/ns/netconf/forward/1.0"
    xmlns:fwd="http://tail-f.com/ns/netconf/forward/1.0"
           xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
    xml:lang="en">

  <!-- <forward> operation -->
  <xs:element name="forward" substitutionGroup="nc:rpcOperation">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="nc:rpcOperationType">
            <xs:sequence>
              <xs:element name="target" type="xs:string"/>
              <xs:element name="auth" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="mechanism" type="xs:string"/>
                    <xs:element name="initial-response" minOccurs="0"
                                type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>


  <!-- <challenge-response> operation -->
```

```
  <xs:element name="challenge-response" substitutionGroup="nc:rpcOperation">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="nc:rpcOperationType">
            <xs:sequence>
              <xs:element name="response" type="xs:string"/>
            </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>

  <!-- reply to <forward> and <challenge-response> operations -->
  <xs:element name="capabilities">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="capability"
                    minOccurs="0" maxOccurs="unbounded"
                    type="xs:uri"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- reply to <forward> and <challenge-response> operations -->
  <xs:element name="challenge" type="xs:string"/>

  <!-- <error-info> content when <error-app-tag> is "sasl-failure" -->
  <xs:element name="failure">
    <xs:complexType>
      <xs:sequence>
        <xs:choice>
          <xs:element name="incorrect-encoding">
            <xs:complexType/>
          </xs:element>
          <xs:element name="invalid-authzid">
            <xs:complexType/>
          </xs:element>
          <xs:element name="invalid-mechanism">
            <xs:complexType/>
          </xs:element>
          <xs:element name="mechanism-too-weak">
            <xs:complexType/>
          </xs:element>
          <xs:element name="not-authorized">
            <xs:complexType/>
          </xs:element>
          <xs:element name="temporary-auth-failure">
            <xs:complexType/>
          </xs:element>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!--
     <error-info> content when <error-app-tag> is
     "sasl-authentication-needed"
  -->
  <xs:element name="mechanisms">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="mechanism"
                    minOccurs="0" maxOccurs="unbounded"
                    type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  <xs:element>


</xs:schema>
```

15.12. Inactive Capability

15.12.1. Overview

This capability is used by the NETCONF server to indicate that it supports marking nodes as being inactive. A node that is marked as inactive exists in the data store, but is not used by the server. Any node can be marked as inactive.

In order to not confuse clients that do not understand this attribute, the client has to instruct the server to display and handle the inactive nodes. An inactive node is marked with an "inactive" XML attribute, and in order to make it active, the "active" XML atribute is used.

This capability is formally defined in the YANG module "tailf-netconf-inactive".

15.12.2. Dependencies

None.

15.12.3. Capability Identifier

The inactive capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/inactive/1.0
```

15.12.4. New Operations

None.

15.12.5. Modifications to Existing Operations

A new parameter, <with-inactive>, is added to the <get>, <get-config>, <edit-config>, <copy-config>, and <start-transaction> operations.

The <with-inactive> element is defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace, and takes no value.

If this parameter is present in <get>, <get-config>, or <copy-config>, the NETCONF server will mark inactive nodes with the "inactive" attribute.

If this parameter is present in <edit-config> or <copy-config>, the NETCONF server will treat inactive nodes as existing, so that an attempt to create a node which is inactive will fail, and an attempt to delete a node which is inactive will succeed. Further, the NETCONF server accepts the "inactive" and "active" attributes in the data hierarchy, in order to make nodes inactive or active, respectively.

If the parameter is present in <start-transaction>, it MUST also be present in any <edit-config>, <copy-config>, <get>, or <get-config> operations within the transaction. If it is not present in <start-transaction>, it MUST NOT be present in any <edit-config> operation within the transaction.

The "inactive" and "active" attributes are defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace. The "inactive" attribute's value is the string "inactive", and the "active" attribute's value is the string "active".

Example

This request creates an inactive interface:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface inactive="inactive">
          <name>Ethernet0/0</name>
          <mtu>1500</mtu>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

This request shows the inactive interface:

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
  </get-config>
</rpc>

<rpc-reply message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <interface inactive="inactive">
        <name>Ethernet0/0</name>
        <mtu>1500</mtu>
      </interface>
    </top>
```

```
  </data>
</rpc-reply>
```

This request shows that inactive data is not returned unless the client asks for it:

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>

<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
  </data>
</rpc-reply>
```

This request activates the interface:

This request creates an inactive interface:

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface active="active">
          <name>Ethernet0/0</name>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

15.13. Tail-f Identification Capability

15.13.1. Overview

This capability is used by a NETCONF peer to inform the other peer about the NETCONF stack and NETCONF client. The receiving peer can use this information in log files etc.

The information a peer may advertise is:

vendor: The vendor of the NETCONF stack.

product: The NETCONF product.

version: The version of the product.

client-identity: The identity of the user starting the session. This parameter can be the local user name of the operator in the client tool.

All these parameters are free form strings, advertised as query parameters to the capability URI, in the <hello> message.

15.13.2. Dependencies

None.

15.13.3. Capability Identifier

The identification capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/identification/1.0
```

15.13.4. Example

This is an example of how a client might advertise its identification information. Whitespace is added to make the example more readable.

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>
      urn:ietf:params:netconf:base:1.1
    </capability>
    <capability>
      http://tail-f.com/ns/netconf/identification/1.0?
             vendor=tail-f
        &amp;product=ncs
        &amp;version=1.8
        &amp;client-identity=admin
    </capability>
  </capabilities>
</hello>
```

15.13.5. ConfD

If a NETCONF client advertises this capability, ConfD picks up the information, and stores it in the user session. The information is available to application programmers through the function maapi_get_user_session_identification().

15.14. The Query API

The Query API consists of a number of RPC operations to start queries, fetch chunks of the result from a query, restart a query, and stop a query.

In the installed release there are two YANG files named `tailf-netconf-query.yang` and `tailf-common-query.yang` that defines these operations. An easy way to find the files is to run the following command from the top directory of release installation:

```
$ find . -name tailf-netconf-query.yang
```

The API consists of the following operations:

- `start-query`: Start a query and return a query handle.
- `fetch-query-result`: Use a query handle to repeatedly fetch chunks of the result.
- `reset-query`: (Re)set where the next fetched result will begin from.
- `stop-query`: Stop (and close) the query.

In the following examples, the following data model is used:

```
container x {
  list host {
    key number;
    leaf number {
      type int32;
    }
    leaf enabled {
      type boolean;
    }
    leaf name {
      type string;
    }
    leaf address {
      type inet:ip-address;
    }
  }
}
```

Here is an example of a `start-query` operation:

```
<start-query xmlns="http://tail-f.com/ns/tailf-netconf-query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <limit>100</limit>
  <offset>1</offset>
</start-query>
```

An informal interpretation of this query is:

For each `/x/host` where `enabled` is true, select its `name`, and `address`, and return the result sorted by `name`, in chunks of 100 results at the time.

Let us discuss the various pieces of this request.

The actual XPath query to run is specified by the `foreach` element. In the example below will search for all `/x/host` nodes that has the `enabled` node set to `true`:

```
<foreach>
  /x/host[enabled = 'true']
</foreach>
```

Now we need to define what we want to have returned from the node set by using one or more `select` sections. What to actually return is defined by the XPath `expression`.

We must also choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk The possible result-types are: `string` , `path` , `leaf-value` and `inline`.

The difference between `string` and `leaf-value` is somewhat subtle. In the case of `string` the result will be processed by the XPath function `string()` (which if the result is a node-set will concatenate all the values). The `leaf-value` will return the value of the first node in the result. As long as the result is a leaf node, `string` and `leaf-value` will return the same result. In the example above, we are using `string` as shown below. At least one `result-type` must be specified.

The result-type `inline` makes it possible to return the full sub-tree of data in XML format. The data will be enclosed with a tag: `data`.

Finally we can specify an optional `label` for a convenient way of labeling the returned data. In the example we have the following:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as XPath expressions, which in most cases are very simple and refers to the found node set. In this example we sort the result by the content of the `name` node:

```
<sort-by>name</sort-by>
```

To limit the max amount of results in each chunk that `fetch-query-result` will return we can set the `limit` element. The default is to get all results in one chunk.

```
<limit>100</limit>
```

With the `offset` element we can specify at which node we should start to receive the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

Now, if we continue by putting the operation above in a file `query.xml` we can send a request, using the command **netconf-console**, like this:

```
$ netconf-console --rpc query.xml
```

The result would look something like this:

```
<start-query-result>
```

```
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example "12345") must be used in all subsequent calls. To retrieve the result, we can now send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/tailf-netconf-query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/tailf-netconf-query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
</query-result>
```

If we try to get more data with the `fetch-query-result` we might get more `result` entries in return until no more data exists and we get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/tailf-netconf-query">
</query-result>
```

If we want to go back in the "stream" of received data chunks and have them repeated, we can do that with the `reset-query` operation. In the example below we ask to get results from the 42:nd result entry:

```
<reset-query xmlns=\"http://tail-f.com/ns/tailf-netconf-query\">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

Finally, when we are done we stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/tailf-netconf-query">
  <query-handle>12345</query-handle>
</stop-query>
```

15.15. Meta-data in Attributes

ConfD supports three pieces of meta-data data nodes: tags, annotations, and inactive.

This feature is by default disabled, but can be enabled by setting `/confdConfig/enableAttributes` to true in `confd.conf` (see confd.conf(5)).

An annotation is a string which acts a comment. Any data node present in the configuration can get an annotation. An annotation does not affect the underlying configuration, but can be set by a user to comment what the configuration does.

An annotation is encoded as an XML attribute 'annotation' on any data node. To remove an annotation, set the 'annotation' attribute to an empty string.

Any configuration data node can have a set of tags. Tags are set by the user for data organization and filtering purposes. A tag does not affect the underlying configuration.

All tags on a data node are encoded as a space separated string in an XML attribute 'tags'. To remove all tags, set the 'tags' attribute to an empty string.

Annotation, tags, and inactive attributes can be present in `<edit-config>`, `<copy-config>`, `<get-config>`, and `<get>`. For example:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <interfaces xmlns="http://example.com/ns/if">
        <interface annotation="this is the management interface"
                   tags=" important ethernet ">
          <name>eth0</name>
          ...
        </interface>
      </interfaces>
    </config>
  </edit-config>
</rpc>
```

15.16. Namespace for Additional Error Information

ConfD adds an additional namespace which is used to define elements which are included in the `<error-info>` element. This namespace also describes which `<error-app-tag/>` elements the server might generate, as part of an `<rpc-error/>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/params/1.1"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xml:lang="en">

  <xs:annotation>
    <xs:documentation>
      Tail-f's namespace for additional error information.
      This namespace is used to define elements which are included
      in the 'error-info' element.

      The following are the app-tags used by the NETCONF agent:

        o  not-writable
```

```
        Means that an edit-config or copy-config operation was
        attempted on an element which is read-only
        (i.e. non-configuration data).

      o  missing-element-in-choice

        Like the standard error missing-element, but generated when
        one of a set of elements in a choice is missing.

      o  pending-changes

        Means that a lock operation was attempted on the candidate
        database, and the candidate database has uncommitted
        changes. This is not allowed according to the protocol
        specification.

      o  url-open-failed

        Means that the URL given was correct, but that it could not
        be opened. This can e.g. be due to a missing local file, or
        bad ftp credentials. An error message string is provided in
        the &lt;error-message&gt; element.

      o  url-write-failed

        Means that the URL given was opened, but write failed. This
        could e.g. be due to lack of disk space. An error message
        string is provided in the &lt;error-message&gt; element.

      o  bad-state

        Means that an rpc is received when the session is in a state
        which don't accept this rpc.  An example is
        &lt;prepare-transaction&gt; before &lt;start-transaction&gt;

  </xs:documentation>
</xs:annotation>

<xs:element name="bad-keyref">
  <xs:annotation>
    <xs:documentation>
      This element will be present in the 'error-info' container when
      'error-app-tag' is "instance-required".
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="bad-element" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Contains an absolute XPath expression pointing to the element
            which value refers to a non-existing instance.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="missing-element" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Contains an absolute XPath expression pointing to the missing
            element referred to by 'bad-element'.
```

```
          </xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="bad-instance-count">
  <xs:annotation>
    <xs:documentation>
      This element will be present in the 'error-info' container when
      'error-app-tag' is "too-few-elements" or "too-many-elements".
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="bad-element" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Contains an absolute XPath expression pointing to an
            element which exists in too few or too many instances.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="instances" type="xs:unsignedInt">
        <xs:annotation>
          <xs:documentation>
            Contains the number of existing instances of the element
            referd to by 'bad-element'.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:choice>
        <xs:element name="min-instances" type="xs:unsignedInt">
          <xs:annotation>
            <xs:documentation>
              Contains the minimum number of instances that must
              exist in order for the configuration to be consistent.
              This element is present only if 'app-tag' is
              'too-few-elems'.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="max-instances" type="xs:unsignedInt">
          <xs:annotation>
            <xs:documentation>
              Contains the maximum number of instances that can
              exist in order for the configuration to be consistent.
              This element is present only if 'app-tag' is
              'too-many-elems'.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:attribute name="annotation" type="xs:string">
  <xs:annotation>
```

```
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        acts as a comment for the node.  The annotation does not affect the
        underlying configuration data.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

  <xs:attribute name="tags" type="xs:string">
    <xs:annotation>
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        is a space separated string of tags for the node.  The tags of a
        node does not affect the underlying configuration data, but can
        be used by a user for data organization, and data filtering.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

</xs:schema>
```

Chapter 16. The CLI agent

16.1. Overview

Confd provides three different CLI styles, one inspired by the Junos CLI (J), one inspired by the Cisco XR CLI (C), and one inspired by the Cisco IOS CLI (I). All styles can be supported at the same time, or one style can be chosen for a given deployment of ConfD. The default style is configured in the `confd.conf` file using the *style* setting in the *cli* section.

The CLI is automatically rendered using the data model described by the *yang* files. This way we get an auto-generated CLI, without any extra effort, except the design of our yang files. The auto-generated CLI supports the following features:

- Command line history and command line editor.
- Tab completion for content of the configuration database.
- Monitoring and inspecting log files.
- Inspecting the system configuration and system state.
- Fully configuring the system.

Alias expansion is performed when a command line is entered. Aliases are part of the configuration and are manipulated accordingly. In the J-style CLI this is done by manipulating the nodes in the alias configuration tree. In the C- and I-style CLIs this is done by the alias command in configuration mode.

The C- and I-style CLIs automatically create modes for each list node in the *yang* files, and commands for setting each parameter are generated. Actions specified in the *yang* files are mapped to commands in the mode where they appear.

The J-style CLI contains commands for manipulating the configuration and actions in the *yang* files are mapped into **request** commands in operational mode.

Even though the auto-generated CLI is fully functional it can be customized and extended in numerous ways:

- Built-in commands can be moved, deleted and reordered.
- Confirmation prompts can be added to built-in commands.
- New commands can be implemented using the C-API and ordinary executables, and shell scripts can be invoked from a command.
- New commands can be mounted freely in the existing command hierarchy.
- The built-in tab completion mechanism can be overridden using user defined callbacks.
- New command hierarchies can be created.
- A command timeout can be added, both a global timeout for all commands, and command specific timeouts.
- Actions and parts of the data tree can be hidden and can later be made visible when the user enters a password.

In the C- and I-style CLIs some additional customizations are possible.

- The automatically generated modes can be suppressed.
- New modes can be added at internal nodes.
- The builtin show output can be replaced with an arbitrary command, either for the whole configuration or just parts of it.
- Custom mode names can be assigned statically or dynamically through a C-callback
- Transactions can be disabled so that all CLI modifications take effect immediately. The IOS style CLI automatically disables transactions.

How to customize and extend the auto-generated CLI is described in the *clispec(5)* manual page.

Tip: In the ConfD distribution there is an Emacs mode suitable for clispec editing.

16.2. The J-style CLI

16.2.1. Command Hierarchy

The CLI is built around a hierarchy of commands. This makes it possible to logically group commands. The operational mode command hierarchy looks like this for the XR CLI:

```
--|- commit
  |- compare -
  |         |- file
  |         |- startup
  |- configure
  |- file ---
  |         |- list
  |         |- show
  |         |- rename
  |         |- delete
  |         |- compare -
  |         |          |- files
  |         |- copy
  |- help
  |- id
  |- monitor -
  |           |
  |           |- stop
  |           |- start
  |- ping
  |- quit
  |- request -
  |           |- <action>
  |           |- job -
  |                    |- stop
  |           |- message
  |           |- system -
  |                      |- logout -
  |                                |- user
  |
  |- set
```

```
  |- set-path
  |- show ---
  |          |- all
  |          |- jobs
  |          |- users
  |          |- status
  |          |- configuration
  |          |- cli
  |          |- cli -
  |          |      |- history
  |          |- log
  |          |- notification
  |          |- parser -
  |          |         |- dump
  |- source
  |- ssh
  |- telnet
  |- traceroute
```

The configure mode command hierarchy looks like this:

```
--|- activate
  |- annotate
  |- commit -
  |         |- confirmed
  |         |- check
  |         |- and-quit
  |- compare -
  |          |- file
  |          |- running
  |- deactivate
  |- delete
  |- edit
  |- exit
  |- help
  |- hide
  |- insert
  |- move
  |- load
  |- quit
  |- rename
  |- revert
  |- rollback
  |- run
  |- save
  |- set
  |- show -
  |       |- parser -
  |       |         |- dump
  |- status
  |- tag -
  |      |- add
  |      |- clear
  |      |- del
  |- top
  |- unhide
  |- up
  |- validate
  |- wizard -
  |- adduser
```

16.2.2. Two CLI modes

The ConfD CLI provides various commands for configuring and monitoring software, hardware, and network connectivity of target devices. The CLI supports two modes: *operational mode*, for monitoring the state of the device; and *configure mode*, for changing the state of the device.

The prompt indicates which mode the CLI is in. When moving from operational mode to configure mode using the **configure** command, the prompt is changed from `user@host>` to `user@host%`. The prompts can be configured using the *`prompt1`* and *`prompt2`* settings in the `confd.conf` file.

For example:

```
joe@io> configure
Entering configuration mode "private"

[ok][2006-06-02 12:31:59]
[edit]
joe@io%
```

16.2.3. Operational mode

Operational mode is the initial mode after successful login to the CLI. It is primarily used for viewing the system status, controlling the CLI environment, monitoring and troubleshooting network connectivity, and initiating the configure mode.

The full list of commands available in operational mode is listed below in the "Operational mode commands" section.

16.2.4. Configure mode

Configure mode can be initiated by entering the **configure** command in operational mode. All changes to the device's configuration are done to a copy of the active configuration, called a *candidate configuration*. These changes do not take effect until a successful **commit** or **commit confirm** command is entered.

The full list of commands available in configure mode is listed below in the "Configure mode commands" section.

16.3. The C- and I-style CLI

The C- and I-style CLI is inspired by the Cisco XR CLI and Cisco IOS CLI. The configuration is manipulated through a series of commands and modes. Each parameter in the *yang* files is represented by a separate command.

The CLI provides various commands for configuring and monitoring software, hardware, and network connectivity of target devices. The CLI supports two modes: *operational mode*, for monitoring the state of the device; and *configure mode*, for changing the state of the device. The *configure mode* consists of a number of sub-modes for manipulating different parts of the configuration, i.e. the mode **aaa authentication users user** is present for configuring user authentication parameters.

The prompt indicates which mode the CLI is in. When moving from operational/EXEC mode to configure mode using the **config** command, the prompt is changed from `host#` to `host(mode)#`. The prompts can be configured using the *`cPrompt1`* and *`cPrompt2`* settings in the `confd.conf` file, and additionally in the IOS style through the prompt setting in the AAA configuration.

For example:

```
joe connected from 127.0.0.1 using console on io
io# config terminal
Entering configuration mode private
io(config)#
```

16.3.1. Operational/EXEC mode

Operational mode is the initial mode after successful login to the CLI. It is primarily used for viewing the system status, controlling the CLI environment, monitoring and troubleshooting network connectivity, and initiating the configure mode.

The full list of commands available in operational mode is listed below in the "Operational mode commands" section.

16.3.2. Configure mode

Configure mode can be initiated by entering the **config** command in operational mode. All changes to the device's configuration are done to a copy of the active configuration, called a *candidate configuration*. These changes do not take effect until a successful **commit** or **commit confirm** command is entered.

The full list of commands available in configure mode is listed below in the "Configure mode commands" section. Additional commands and modes are dynamically derived from the *yang* files.

16.4. The CLI in action

16.4.1. Starting the CLI

The CLI is started using the **confd_cli** program. It can either be used as a login program or started manually once the user has logged in.

In a typical device, ordinary users would have the **confd_cli** program as login shell, and the root user would have to login and then start the CLI using **confd_cli**.

confd_cli is a fairly small program (about 800 lines of C code) which we distribute both as a binary and source code (in `$CONFD_DIR/src/confd/cli`). When started it sets the terminal in raw mode, connects to the ConfD daemon through a socket over the loopback interface, and sends user name, the user's groups, protocol, client IP address, terminal settings etc., and then enters a proxying mode where it sends key strokes from the user and prints characters sent by the ConfD daemon.

It is straightforward to modify the C program to send, for example, custom group information. The default behavior is to send the UNIX groups the user belongs to.

Out of the box, the **confd_cli** program supports a range of options, primarily intended for debug and development purposes.

The **confd_cli** program can also be used for batch processing of CLI commands, either by storing the commands in a file and running **confd_cli** on the file, or by having the following line at the top of the file (with the location of the program modified appropriately):

```
#!/bin/confd_cli
```

When the CLI is run non-interactively it will terminate at the first error and will only show the output of the commands executed. It will not output the prompt or echo the commands. This is the same behavior as for shell scripts.

If you want to run a script non-interactively, e.g. as a script or through a pipe, and still want the prompts and commands echoed, you can give the **--interactive** option.

Command line options:

```
confd_cli --help
Usage: confd_cli [options] [file]
Options:
--help, -h            display this help
--host, -H <host>     current host name (used in prompt)
--address, -A <addr>  cli address to connect to
--port, -P <port>     cli port to connect to
--cwd,  -c <dir>      current working directory
--proto, -p <proto>   type of connection (tcp, ssh, console)
--verbose, -v         verbose output
--ip, -i              clients source ip
--interactive, -n     force interactive mode
--juniper, -J         Juniper style CLI
--cisco, -C           Cisco XR style CLI
--user, -u <user>     clients user name
--uid, -U <uid>       clients user id
--groups, -g <groups> clients group list
--gids, -D <gids>     clients group id list
--gid, -G <gid>       clients group id
--noaaa               disable AAA
```

host — The argument to host should be the host name of the device. The **confd_cli** program will use the result of the system call `gethostname()` as default value. The host name is used in the CLI prompt.

address — If ConfD has been configured to listen to a different address than 127.0.0.1 for the communication between subsystems, then that address should be given as argument to `address` (or we can use the `CONFD_IPC_ADDRESS` environment variable, or recompile the **confd_cli** program with the new address compiled in).

port — If ConfD has been configured to use a non-default port for the communication between subsystems, then that port number should be given as argument to `port` (or we can use the `CONFD_IPC_PORT` environment variable, or recompile the **confd_cli** program with the new port compiled in).

cwd — Directory to use as current working directory in the CLI. Normally the user's home directory. The default is the directory where the **confd_cli** program is started.

proto — Should be the protocol used by the user to connect to the box, one of **tcp**, **ssh**, and **console**. The default is **ssh** for connections established with OpenSSH (the program inspects the SSH_CONNECTION environment variable), and **console** for everything else. This value is printed in the audit logs.

verbose — If this argument is given, then the **confd_cli** program will be a bit more talkative during the ConfD handshake phase.

ip — Should be the user's source IP address, if the user connects through SSH or telnet. The default is 127.0.0.1 to indicate the console. This value is printed in the audit logs.

interactive — Force the CLI to echo commands and prompts even when not invoked from a terminal, i.e. when reading input from a file or through a pipe.

cisco — Force the CLI to provide a I-style CLI.

juniper	Force the CLI to provide a Juniper style CLI.
user	The name of the user connecting. Used to set proper access rules and assign proper groups (if the group mapping is kept in ConfD). The default is to use the login name of the user.
uid	The numeric user id of the connected user. The uid will be used when executing osCommands, when checking file access permissions, and when creating files. Note that ConfD needs to run as root for this to work properly.
gid	The numeric group id of the connected user. The gid will be used when executing osCommands, when checking file access permissions, and when creating files. Note that ConfD needs to run as root for this to work properly.
groups	The argument to groups should be a comma-separated list of groups. The default is to send the OS groups that the user belongs to, i.e. the same as the **groups** shell command gives us.
gids	The argument to gids should be a comma-separated list of numeric group ids representing the Unix supplementary groups for the user. These are used when executing osCommands and when checking file access permissions.
noaaa	Disables AAA. This is useful during development but should be removed in a production system.

16.4.2. Protocol between the confd_cli program and ConfD

The initial header sent by the **confd_cli** program to ConfD looks like this:

```
username;source;proto;[OPAQUE=data,]groups;hostname;ttyname;cwd;term;SSH_CONNECTION;style;u
```

username	is the name of the authenticated user.
source	is the textual representation of the ipv4 or ipv6 address which the user connected from (e.g. "10.0.0.1").
proto	is the name of the transport protocol the client used (e.g. "ssh", "tcp", or "console").
data	is an opaque string that can optionally be provided. The data is available to commands as the variable $(opaque). Commas are sent as the NUL (0) character.
groups	is a comma-separated list of group names for the user. This list should only be sent if the transport has the capability to determine which groups a user belongs to. If not, an empty list should be sent. In this case, the normal AAA mechanisms are used to determine group membership.
hostname	is available as $(host) in format strings.
ttyname	is available as $(tty) in command arguments.
cwd	is the current working directory for the user. All relative file accesses will be performed with this directory as base.

term	is the name of the terminal type, ie "vt100".
SSH_CONNECTION	is the contents of the SSH_CONNECTION environment variable. It is available as $(ssh_connection) for command arguments.
style	is one of "j", "i", and "c".
uid, *gid*, and *supgids*	are the UNIX user id, group id, and supplementary group ids for this user. These parameters are used when accessing files and when executing external programs.
width	is a 32-bit integer (little endian) determining the width (in characters) of the terminal.
height	is a 32-bit integer (little endian) determining the height (in characters) of the terminal.
interactive	is a 32-bit integer (little endian), 1 indicates an interactive CLI session, and 0 indicates a non-interactive CLI session.
noaaa	is a 32-bit integer (little endian), 1 indicates that AAA should be disabled for the session, 0 indicates that normal AAA should be performed.

After the initial header characters are proxied to confd with some quoting. The NUL (0) character is quoted with a NUL (0) character. Changes in the terminal size are sent as:

```
<0/8><1/8><width/32><height/32>
```

16.4.3. Starting the CLI in an overloaded system

If the number of ongoing sessions have reached the configured system limit, no more CLI sessions will be allowed until one of the existing sessions have been terminated.

This makes it impossible to get into the system. A situation which may not be acceptable. The CLI therefore has a mechanism for handling this problem. When the CLI detects that the session limit has been reached it will check if the new user has privileges to execute the logout command. If the user does it will display a list of the current user sessions on the box and ask the user if one of the sessions should be terminated to make room for the new session.

16.5. Environment for OS command execution

All OS commands, i.e. executables or shell scripts, are executed using a program called **cmdptywrapper**. To be able to execute an os-command as root or a specific user we need to make **cmdptywrapper** setuid root, i.e.

```
# chown root cmdptywrapper
# chmod u+s cmdptywrapper
```

Failing that, all programs will be executed as the user who started the ConfD daemon. Consequently, if that user is root we do not have to perform the **chmod** operation above.

16.6. Command output processing

It is possible to process the output from a command using an output redirect. This is done using the | character. This redirect feature is supported in both the C- and I-style and the J-style CLI. Most of the

commands are the same but some commands differ. The redirect targets (pipe commands) can be modified using the clispec file, just as a regular CLI command. The commands can be chained to achieve more complex processing.

In the J-style CLI the commands are called - append, count, display set (show configuration only), display annotations, show tags, hide annotations, hide tags, except (exclude), extended, find (begin), linnum, match (include), match-all, match-any, more, nomore, notab (auto-rendered show commands only), repeat (auto-rendered show commands only), save, tab (show commands only) and until. It may look look this in the J-style CLI:

```
admin@tellus> show configuration | ?
Possible completions:
  annotation  - Show only statements whose annotation matches a pattern
  append      - Append output text to a file
  best-effort - Display data even if data provider is unavailable or continue loading from
  count       - Count the number of lines in the output
  details     - Display details
  display     - Display options
  except      - Show only text that does not matches a pattern
  extended    - Show referring elements
  find        - Search for the first occurrence of a pattern
  hide        - Hide display options
  linnum      - Enumerate lines in the output
  match       - Show only text that matches a pattern
  match-all   - All selected filters must match
  match-any   - At least one filter must match
  more        - Paginate output
  nomore      - Suppress pagination
  save        - Save output text to a file
  select      - Select additional columns
  sort-by     - Select sorting indices
  tab         - Enforce table output
  tags        - Show only statements whose tags matches a pattern
  until       - Display until the first occurrence of a pattern
```

In the C- and I-style CLIs the commands are called - append, count, exclude (except), display annotations, display tags, hide annotations, hide tags, begin (find), include (match), linnum, match-all, match-any, more, nomore, notab (auto-rendered show commands only), repeat (auto-rendered show commands only), save, tab (show commands only) and until. It may look look this in the C- and I-style CLI:

```
tellus# show running-config | ?
Possible completions:
  annotation   Show only statements whose annotation matches a pattern
  append       Append output text to a file
  begin        Begin with the line that matches
  count        Count the number of lines in the output
  details      Display commit progress
  display      Display options
  exclude      Exclude lines that match
  extended     Display referring entries
  hide         Hide display options
  include      Include lines that match
  linnum       Enumerate lines in the output
  match-all    All selected filters must match
  match-any    At least one filter must match
  more         Paginate output
  nomore       Suppress pagination
  save         Save output text to a file
  select       Select additional columns
```

```
  sort-by      Select sorting indices
  tab          Enforce table output
  tags         Show only statements whose tags matches a pattern
  until        End with the line that matches
```

The **show annotations/tags** and **hide annotations/tags** pipe targets are only available when viewing the configuration, and only if attributes have been enabled in the confd.conf file.

16.6.1. Sort the the Output

The **sort-by** target makes it possible for the CLI user to control in which order instances should be displayed, and can be used when the path points to a list. The argument to **sort-by** can either be a secondary index or an arbitrary set of leafs in the list. If a secondary index is given as an argument, the table will be sorted in the order defined by the secondary index. If a set of leafs is given as an argument, the table will be sorted in the order in which the leafs are entered. For example:

```
admin@io 13:28:07> show configuration server | sort-by port ip | tab

NAME  IP        PORT  DESCRIPTION
-----------------------------------
1     1.1.1.1   1010  -
7     1.1.1.17  1020  -
10    1.1.1.11  1040  -
3     1.1.1.3   1070  -
6     1.1.1.4   1070  -
5     1.1.1.5   1070  -
4     1.1.1.7   1070  -
8     1.1.1.8   1070  -
9     1.1.1.9   1070  -
11    1.1.1.10  1070  -
2     1.1.1.12  1070  -

[ok][2013-08-31 13:49:44]
admin@io 13:50:12>
```

16.6.2. Count the Number of Lines in the Output

This redirect target counts the number of lines in the output. For example:

```
admin@io 13:28:07> show configuration | count
[ok][2007-08-31 13:49:44]
Count: 99 lines
admin@io 13:49:44> show configuration aaa | count
[ok][2007-08-31 13:50:12]
Count: 90 lines
admin@io 13:50:12>
```

16.6.3. Search for a String in the Output

The **match** target (**include** in C- and I- style) is used to only include lines matching a regular expression. For example:

```
admin@io 13:53:59> show configuration aaa | match {
aaa {
    authentication {
        users {
            user admin {
```

```
            user oper {
            user private {
            user public {
        groups {
            group admin {
            group oper {
    authorization {
        cmdrules {
            cmdrule 1 {
            cmdrule 2 {
            cmdrule 3 {
            cmdrule 150 {
        datarules {
            datarule 101 {
            datarule 203 {
```

In the example above only lines containing { are shown. Similarly lines not containing a regular expression can be included. This is done using the **except** target (**exclude** in C- and I- style). For example:

```
admin@io 13:56:30> show configuration aaa authentication | except {
            uid        1000;
            gid        100;
            password   $1$fB$0w68PmacQ4VmE3/M3nK3Ug==;
            ssh_keydir /var/confd/homes/admin/.ssh;
            homedir    /var/confd/homes/admin;
        }
            uid        1000;
            gid        100;
            password   $1$S6$brGZW9wSDifHoU7Rf5KSHA==;
            ssh_keydir /var/confd/homes/oper/.ssh;
            homedir    /var/confd/homes/oper;
        }
            uid        1000;
            gid        100;
            password   $1$L4$YcCoIivO4mrzoj8vCrEjlw==;
            ssh_keydir /var/confd/homes/private/.ssh;
            homedir    /var/confd/homes/private;
        }
            uid        1000;
            gid        100;
            password   $1$Ft$9zTEc79NWFE0E8v7I2RxVQ==;
            ssh_keydir /var/confd/homes/public/.ssh;
            homedir    /var/confd/homes/public;
        }
    }
            users "admin private";
        }
            users "oper public";
        }
    }
}
```

It is also possible to display the output starting at the first match of a regular expression, using the **find** target (**begin** in C- and I- style). For example:

```
admin@io 14:03:44> show configuration aaa authentication users | find private
user private {
    uid        1019;
    gid        1013;
    password   $1$AO$hbQEgdGQLzlWhX/1FNL5f.;
```

```
    ssh_keydir /var/confd/homes/private/.ssh;
    homedir    /var/confd/homes/private;
}
user public {
    uid        1019;
    gid        1013;
    password   $1$Kh$0Lor2g1yrSQ7MYDLxFr9h0;
    ssh_keydir /var/confd/homes/public/.ssh;
    homedir    /var/confd/homes/public;
}
```

Output can also be ended when a line matches a regular expression. This is done with the **until** target. For example:

```
admin@io 14:03:44> show configuration aaa authentication users | find private | until publi
user private {
    uid        1019;
    gid        1013;
    password   $1$AO$hbQEgdGQLzlWhX/1FNL5f.;
    ssh_keydir /var/confd/homes/private/.ssh;
    homedir    /var/confd/homes/private;
}
user public {
```

It is also possible to filter the output by using a sequence of **select** statements followed by **match-any** or **match-any**. Consider the configuration:

```
admin@io 14:03:44> show configuration servers server
server a {
    ip   1.2.3.4;
    port 23;
}
server b {
    ip   2.3.4.5;
    port 24;
}
server c {
    ip   3.4.5.6;
    port 25;
}
```

If we were to show all servers that has either ip 1.2.3.4 *or* port 24, this can be done by using select statements, like so

```
admin@io 14:03:44> show configuration servers server | select ip 1.2.3.4 | select port 24 |
server a {
    ip   1.2.3.4;
    port 23;
}
server b {
    ip   2.3.4.5;
    port 24;
}
```

whereas a **match-all** filtering would in this case result in

```
admin@io 14:03:44> show configuration servers server | select ip 1.2.3.4 | select port 24 |
No entries found.
```

as there are no servers that has both ip 1.2.3.4 *and* port 24.

16.6.4. Saving the Output to a File

The output can also be saved to a file using the **save** or **append** redirect target. For example:

```
admin@io 14:03:51> show configuration aaa  | save /tmp/saved
```

Or to save the configuration, except all passwords

```
admin@io 14:03:51> show configuration | except password | save /tmp/saved
```

16.6.5. Regular expressions

The regular expressions is a subset of the regular expressions found in egrep and in the AWK programming language. Some common operators are:

.	Matches any character.
^	Matches the beginning of a string.
$	Matches the end of a string.
[abc...]	Character class, which matches any of the characters abc... Char- acter ranges are specified by a pair of characters separated by a -.
[^abc...]	negated character class, which matches any character except abc....
r1 | r2	Alternation. It matches either r1 or r2.
r1r2	Concatenation. It matches r1 and then r2.
r+	Matches one or more rs.
r*	Matches zero or more rs.
r?	Matches zero or one rs.
(r)	Grouping. It matches r.

For example, to only display uid and gid you can do the following:

```
admin@io 15:11:24> show configuration | match "(uid)|(gid)"
                uid        1000;
                gid        100;
                uid        1000;
                gid        100;
                uid        1000;
                gid        100;
                uid        1000;
                gid        100;
```

16.6.6. Display line numbers

The **linnum** target causes a line number to be displayed at the beginning of each line in the display.

```
admin@io 15:11:24> show configuration | match "(uid)|(gid)" | linnum
1:                uid        1019;
2:                gid        1013;
3:                uid        1019;
4:                gid        1013;
```

```
5:                 uid        1019;
6:                 gid        1013;
7:                 uid        1019;
8:                 gid        1013;
```

16.7. Range expressions

It is possible to modify a range of instances at the same time using range expressions, and to display a range of instances using a range expression.

Key attributes that are integers are automatically support range expressions, both in the J- style and the I- and C- style CLI. The syntax is slightly different in the J-style than in I- and C- style.

Automatic range expressions are also supported for key elements of other types as long as they are restricted to the pattern [a-zA-Z-]*[0-9]+/[0-9]+/[0-9]+/.../[0-9]+. I.e., the CLI understands the [integer]/[integer] syntax.

If you have more complicated data-types then you may need to write a custom range callback. See the `examples.confd/cli/custom_range` example.

Suppose we have data model like this:

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

  import ietf-inet-types {
    prefix inet;
  }

  import tailf-common {
    prefix tailf;
  }

  typedef interface-type {
    type string {
      pattern "((FastEthernet-)|(GigaEthernet-))[0-9]+/[0-9]+/[0-9]+";
    }
  }

  list server {
    key name;
    max-elements 64;
    leaf name {
      type int32;
    }
    leaf ip {
      type inet:ip-address;
      mandatory true;
    }
    leaf port {
      type inet:port-number;
      mandatory true;
    }
    leaf description {
      type string;
    }
```

```
  }

  list interface {
    key name;
    max-elements 64;
    leaf name {
      type interface-type;
    }
    leaf ip {
      type inet:ip-address;
      mandatory true;
    }
    leaf mtu {
      type int32;
      mandatory true;
    }
    leaf description {
      type string;
    }
    tailf:action reset {
        tailf:actionpoint reset-point;
        input {
            leaf mode {
                type string;
                mandatory true;
            }
            leaf debug {
                type empty;
            }
        }
        output {
            leaf time {
                type string;
                mandatory true;
            }
        }

    }
  }
}
```

Then you can do the following in the J-style CLI. Display a selected subset of the servers:

```
show configuration server 1-3,5
```

Display the configuration of a subset of the interfaces:

```
show configuration interface FastEthernet-1/1/1,2
```

In configure mode you can edit a range of interfaces in one go. For example:

```
jb@io> configure
Entering configuration mode private
[ok][2009-06-16 12:57:59]

[edit]
jb@io% edit interface FastEthernet-1/1/1-3
[ok][2009-06-16 12:58:14]

[edit interface FastEthernet-1/1/1-3]
```

```
jb@io% set mtu 1400
[ok][2009-06-16 12:58:20]

[edit interface FastEthernet-1/1/1-3]
jb@io% commit
Commit complete.
[ok][2009-06-16 12:58:22]

[edit interface FastEthernet-1/1/1-3]
jb@io%
```

In the C- and I- style CLIs the syntax is slightly different in configure mode. When you want to modify a range of instances you enter a specific range mode.

```
io# config
Entering configuration mode terminal
io(config)# interface range FastEthernet-1/1/1-3
io(config-interface-FastEthernet-1/1/1-3)# mtu 1300
io(config-interface-FastEthernet-1/1/1-3)# commit
Commit complete.
io(config-interface-FastEthernet-1/1/1-3)#
```

See the `examples.confd/cli/range` example.

16.8. Autorendering of enabled/disabled

If the data model contains an element called `enabled` of type xs:boolean then it will get some special treatment. Normally the user would have to enter **enabled true** and **enabled false** to enable/disable. To make this a bit more user friendly the CLI will also accept **enabled** and **disabled**. A **disabled** command will be auto generated if there is an **enabled** option of the proper type.

However, the **disabled** command will not be visible in configuration dumps, instead **enabled false** will be shown. The **disabled** command is only provided as a sugar in the CLI.

This behaviour can be controlled on a global level by configuring the `confd.conf` setting */confdConfig/cli/useShortEnabled*. If the value is set to 'true', it is possible to disable the behavior on a per leaf basis, by using the extension *tailf:cli-suppress-shortenabled*.

16.9. Actions

Actions are invoked differently in C/I- and J-mode. In C/I-mode an action appears as a mode specific command, whereas in J-mode an action is invoked using the **request** command in operational mode.

For example, given the data model fragment below:

```
tailf:action shutdown {
  tailf:actionpoint actions;
  input {
    tailf:constant-leaf flags {
      type uint64 {
        range "1 .. max";
      }
      tailf:constant-value 42;
    }
    leaf timeout {
      type xs:duration;
      default PT60S;
```

```
    }
    leaf message {
      type string;
    }
    container options {
      leaf rebootAfterShutdown {
        type boolean;
        default false;
      }
      leaf forceFsckAfterReboot {
        type boolean;
        default false;
      }
      leaf powerOffAfterShutdown {
        type boolean;
        default true;
      }
    }
  }
}
```

In J-mode the restart action is invoked as follows:

```
joe@io> request shutdown timeout 10s message reboot options { forceFsckAfterReboot true }
```

And in C/I-mode as a mode specific command as follows:

```
io(config)# system restart mode quick restore { db startup  fast } debug timeout 45
```

Full command and argument completion is available when entering the command.

The order in which the action arguments are entered is not important. The parameters are reordered by the CLI-backend before invoking the action callback.

16.10. Command history

Command history is maintained in each mode. When we enter configure mode, we will get an empty history, i.e. we cannot access the command history from operational mode. When we exit back into operational mode we will again have access to the command history from the preceding operational mode session.

16.11. Command line editing

The default key strokes for editing the command line and moving around the command history are as follows. Note that it is possible to change these commands using the keymap clispec modification. See the clispec.5 man page for more details.

16.11.1. Moving the cursor:

Move the cursor back one character
: Ctrl-b or Left Arrow

Move the cursor back one word
: Esc-b or Alt-b

Move the cursor forward one character
: Ctrl-f or Right Arrow

Move the cursor forward one word
Esc-f or Alt-f

Move the cursor to the beginning of the command line
Ctrl-a or Home

Move the cursor to the end of the command line
Ctrl-e or End

16.11.2. Delete characters:

Delete the character before the cursor
Ctrl-h, Delete, or Backspace

Delete the character following the cursor
Ctrl-d

Delete all characters from the cursor to the end of the line
Ctrl-k

Delete the whole line
Ctrl-u or Ctrl-x

Delete the word before the cursor
Ctrl-w, Esc-Backspace, or Alt-Backspace

Delete the word after the cursor
Esc-d or Alt-d

16.11.3. Insert recently deleted text:

Insert the most recently deleted text at the cursor
Ctrl-y

16.11.4. Display previous command lines:

Scroll backward through the command history
Ctrl-p or Up Arrow

Scroll forward through the command history
Ctrl-n or Down Arrow

Search the command history in reverse order
Ctrl-r

Show a list of previous commands
run the "show cli history" command

16.11.5. Capitalization:

Capitalize the word at the cursor, i.e. make the first character uppercase and the rest of the word lowercase.
Esc-c

Change the word at the cursor to lowercase.
Esc-l

Change the word at the cursor to uppercase.
Esc-u

16.11.6. Special:

Abort a command/Clear line
Ctrl-c

Quote insert character, i.e. do not treat the next keystroke as an edit command.
Ctrl-v/ESC-q

Redraw the screen
Ctrl-l

Transpose characters
Ctrl-t

Enter multi-line mode. This lets you enter multi-line values when prompted for a value in the CLI. It is not available when editing a CLI command.
ESC-m

Exit configuration mode. Only in C- and I-style.
Ctrl-z

16.12. Using CLI completion

We do not always have to type the full command or option name for the CLI to recognize it. To display possible completions, type the partial command followed immediately by <tab> or <space>.

If the partially typed command uniquely identifies a command, the full command name will appear. Otherwise a list of possible completions is displayed.

Completion is disabled inside quotes; in other words, if we want to type an argument containing spaces, we can either quote them with a \ (e.g. **show file foo\ bar**) or with a " (e.g. **show file "foo bar"**). Space completion is disabled when entering a filename.

Command completion also applies to filenames and directories.

Example:

```
admin@io> <space>
Possible completions:
commit     - Confirm a pending commit
configure  - Manipulate software configuration information
file       - Perform file operations
help       - Provide help information
monitor    - Real-time debugging
ping       - Ping a host
quit       - Exit the management session
request    - Make system-level requests
set        - Set CLI properties
set-path   - Set relative show-path
show       - Show information about the system
ssh        - Open a secure shell on another host
telnet     - Open a telnet session to another host
traceroute - Trace the route to a remote host
admin@io> re<space>quest <space>
```

```
Possible completions:
job     - Job operations
message - Send message to terminal of one or all users
system  - System operations
admin@io> request  m<space>essage all "hello"
```

16.12.1. Customizing CLI completion

CLI completion kicks in for both command parameters and for values adhering to leaf elements in yang. By default completions are generated automatically depending on the parameter or leaf element type. Some examples:

```
d199# history ?
<size>
```

Above we trigger a ?-completion for the built-in **history** command which takes a single parameter of type *xs:nonNegativeInteger*. The built-in history command chooses to answer with a <*size*> completion string.

```
d199# config
Entering configuration mode terminal
d199(config)# interface ?
Possible completions:
  <name: GigaEthernetX/Y>
```

Above we trigger a ?-completion for the list element *interface* which has a key leaf element of type *interfaceNameType* defined in the *http://tail-f.com/ns/example/config/1.0* namespace:

```
  typedef interfaceNameType {
    tailf:info "GigaEthernetX/Y";
    type string;
  }
```

If the default completion behavior is not satisfactory a custom completion callback can be used instead. This holds true for both built-in and user defined commands as well for leaf elements in built-in and user defined data models.

In the following examples we redefined the completion behavior for the two examples above.

```
d199# history ?
Possible completions:
  500  750  The history must be a non-negative value (preferably 500 or 750)
```

and

```
d199# config
Entering configuration mode terminal
d199(config)# interface ?
Possible completions:
  FastEthernet0/1   FastEthernet IEEE 802.3
  GigaEthernet0/1   GigabitEtherenet IEEE 802.3z
  GigaEthernet0/2   GigabitEtherenet IEEE 802.3z
  GigaEthernet1/1   GigabitEtherenet IEEE 802.3z
  GigaEthernet1/2   GigabitEtherenet IEEE 802.3z
d199(config)# interface GigaEthernet ?
Possible completions:
  GigaEthernet0/1  GigaEthernet0/2  GigaEthernet1/1  GigaEthernet1/2
```

This was achieved using the following clispec fragment:

```
    <modifications>
```

```
      <simpleType namespace="" name="uint64">
        <callback>
          <capi>
            <completionpoint>generic-complete</completionpoint>
          </capi>
        </callback>
      </simpleType>
      <simpleType namespace="http://tail-f.com/ns/example/config"
                  name="interfaceNameType">
        <callback>
          <capi>
            <completionpoint>ifs-complete</completionpoint>
          </capi>
        </callback>
      </simpleType>
```

The *generic-complete* and *ifs-complete* callbacks look like this (fragments):

```
static int generic_complete(struct confd_user_info *uinfo, int cli_style,
                            char *token, int completion_char,
                            confd_hkeypath_t *kp, char *cmdpath,
                            char *cmdparam_id,
                            struct confd_qname *simpleType, char *extra) {
    char keypath[BUFSIZ] = {0};
    struct confd_completion_value values[6];
    int i = 0;


    if (strcmp(cmdpath, "history") == 0 ||
        strcmp(cmdpath, "set history") == 0) {
        values[i].type = CONFD_COMPLETION_INFO;
        values[i].value = "The history must be a non-negative value (preferably 500 or 750)
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "500";
        values[i].extra = NULL;
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "750";
        values[i].extra = NULL;
        i++;
    }


    if (confd_action_reply_completion(uinfo, values, i) < 0)
        confd_fatal("Failed to reply to confd\n");

    return CONFD_OK;
}


static int ifs_complete(struct confd_user_info *uinfo, int cli_style,
                        char *token, int completion_char, confd_hkeypath_t *kp,
                        char *cmdpath, char *cmdparam_id,
                        struct confd_qname *simpleType, char *extra) {
    char keypath[BUFSIZ] = {0};
    struct confd_completion_value values[6];
    int i = 0;
```

```
    if (completion_char == '?') {
        values[i].type = CONFD_COMPLETION;
        values[i].value = "GigaEthernet0/1";
        values[i].extra = "GigabitEtherenet IEEE 802.3z";
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "GigaEthernet0/2";
        values[i].extra = "GigabitEtherenet IEEE 802.3z";
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "GigaEthernet1/1";
        values[i].extra = "GigabitEtherenet IEEE 802.3z";
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "GigaEthernet1/2";
        values[i].extra = "GigabitEtherenet IEEE 802.3z";
        i++;

        values[i].type = CONFD_COMPLETION;
        values[i].value = "FastEthernet0/1";
        values[i].extra = "FastEthernet IEEE 802.3";
        i++;
    }


    if (confd_action_reply_completion(uinfo, values, i) < 0)
        confd_fatal("Failed to reply to confd\n");

    return CONFD_OK;
}
```

Take a look at the completion callback example which comes bundled with the ConfD distribution, i.e. `cli/completions` and read more about the completion callback API in the confd_lib_dp(3) manual page.

16.13. Using the comment characters # or !

All characters following a # (in J-style) or ! (in C-style) character up to the next newline are ignored. This makes it possible to have comments in a file containing CLI commands, and still be able to paste the file into the command-line interface. For example:

```
# Command file create 2006-05-20 by Joe Smith
# First show the configuration before we change it
show configuration
# Enter configuration mode and add joe as user
configure
wizard adduser
joe
foobar
foobar
admin
commit
exit
```

```
# Done
```

If we want to enter the # or the ! character as an argument, it has to be prefixed with a backslash (\) or used inside quotes (").

16.14. Annotations and tags

If you have large configurations it may make sense to be able to associate comments (annotations) and tags with the different parts. And then be able to filter the configuration with respect to the annotations or tags. For example, tagging parts of the configuration that relates to a certain department or customer.

ConfD has support for both tags and annotations. The support is enabled by enabling configuration attributes in the `confd.conf` file. I.e.,

```
  <enableAttributes>true</enableAttributes>
```

Once attributes has been enabled a set of new commands will be available in the CLI for annotating and tagging parts of the configuration. There will also be a set of pipe targets for controlling whether the tags and annotations should be displayed, and for filtering depending on annotation and tag content.

The new commands are:

- **annotate <statement> <text>**
- **tag add <statement> <tag>**
- **tag del <statement> <tag>**
- **tag clear <statement> <tag>**

The annotations and tags will be displayed as comments where the tags are prefixed by **Tags:**. For example:

```
joe@io 16:10:17% annotate aaa authentication users user admin "Only allow the XX department
[ok][2009-09-29 16:17:16]

[edit]
joe@io 16:17:16% tag add aaa authentication users user oper foo
[ok][2009-09-29 16:28:28]

[edit]
joe@io 16:29:02% show aaa authentication users user | tags foo
/* Tags: foo */
user oper {
    uid        1000;
    gid        1000;
    password   $1$mfy4jdVt$dNJbiaylcbjpNIeRvHs3X0;
    ssh_keydir /var/confd/homes/oper/.ssh;
    homedir    /var/confd/homes/oper;
}
[ok][2009-09-29 16:29:18]

[edit]
joe@io 16:29:29% show aaa authentication users user | annotation *XX*
/* Only allow the XX department access to this user. */
user admin {
    uid        1000;
    gid        1000;
```

```
    password   $1$Qe$71aKksCOyR.KTBG6ojcGg1;
    ssh_keydir /var/confd/homes/admin/.ssh;
    homedir    /var/confd/homes/admin;
}
[ok][2009-09-29 16:29:33]

[edit]
joe@io 16:29:33%
```

It is possible to hide the tags and annotations when viewing the configuration, or to explicitly include them in the listing. This is done using the **display annotations/tags** and **hide annotations/tags** pipe targets.

Note that annotations are tags are part of the configuration. If you add, remove or modify an annotation or tag you need to commit the new configuration, just as you would if you have made any other change to the configuration. In I-style this will happen automatically once you press *enter*.

See the `examples.confd/cli/annotations` example for a hands on experience.

16.15. Activate and Deactivate

It may be useful to be able to deactivate parts of a configuration without actually removing it from the configuration file. It makes it possible to, for example, pre-provision a device or temporarily disable parts of a configuration without loosing the config.

ConfD has support for activate/deactivate. The support is enabled by enabling configuration attributes in the `confd.conf` file. I.e.,

```
  <enableInactive>true</enableInactive>
```

Once inactive has been enabled two new commands becomes available in configure mode: **activate** and **deactivate**. The argument to the command is a path into the configuration with the same properties as a path that is deletable.

The new commands are:

- **activate <statement>**
- **deactivate <statement>**

A deactivated statement will be indicated by a comment that says *Inactive* on a line before the deactivated statement in C- and I-style, and by an *inactive:* prefix in J-style.

In the J-style CLI

```
joe@io% deactivate server 1
[ok][2010-11-02 14:45:25]

[edit]
joe@io% show server
inactive: server 1 {
    ip   1.1.1.1;
    port 1071;
}
server 2 {
    ip   1.1.1.2;
    port 1072;
```

```
}
server 3 {
    ip   1.1.1.3;
    port 1073;
}
[ok][2010-11-02 14:45:33]

[edit]
joe@io%
```

In the C- and I-style CLIs

```
io(config)# deactivate server 1
io(config)# show config
/* Inactive */
server 1
!
io(config)# show configuration merge server
/* Inactive */
server 1
 ip   1.1.1.1
 port 1071
!
server 2
 ip   1.1.1.2
 port 1072
!
server 3
 ip   1.1.1.3
 port 1073
!
io(config)#
```

See the `examples.confd/cli/annotations` example for a hands on experience.

16.16. CLI messages

Messages appear when we enter and exit configure mode, when we commit a configuration, and when we type a command or value that is not valid.

Examples:

```
admin@io> show c
----------------^
syntax error: unknown command
Possible commands starting with c:
configuration - Display current configuration
cli           - Display cli settings
admin@io> show configuration
------------------^
syntax error: unknown command
[error][2006-06-09 10:12:05]

admin@io% set aaa auth
----------------------^
syntax error: element does not exist
Possible  values starting with auth:
authentication
```

```
authorization
```

When we commit a configuration, the CLI first validates the configuration and if there is a problem indicates what the problem is, for example a missing identifier or a value out of range. A message indicates where the errors are.

Examples:

```
admin@io> configure
Entering configuration mode "private"
[ok][2006-06-02 12:31:59]

[edit]
admin@io% set aaa authorization rules rule 200
[ok][2006-06-02 12:31:59]

[edit]
admin@io% commit
Aborted: value is unset 'aaa authorization rules rule 200 action'
[error][2006-06-02 12:31:59]

[edit]
admin@io%
```

16.17. confd.conf settings

Parts of the CLI behavior can be controlled from the `confd.conf` file. See the confd.conf.5 man page for a comprehensive description of all the options.

16.18. CLI Environment

There are a number of session variables in the CLI. They are only used during the session and are not persistent. Their values are inspected using **show cli** in operational mode, and set using **set** in operational mode. Their initial values are in order derived from the content of the confd.conf file, and the global defaults as configured under /aaa:session, and finally from user specific settings configured under under /aaa:user{<user>}/setting.

```
admin@io> show cli
cli {
    autowizard true;
    complete-on-space true;
    ignore-leading-space false;
    history 100;
    idle-timeout 1800;
    output {
        file terminal;
    }
    paginate true;
    screen {
        length 82;
        width 80;
    }
    show {
        defaults false;
    }
    terminal xterm;
}
```

```
[ok][2006-06-02 12:31:59]
admin@io>
```

The different values control different parts of the CLI behavior.

autowizard (*true* | *false*)
When enabled, the CLI will prompt the user for required settings when a new identifier is created and for mandatory action parameters.

For example:

```
admin@io% set aaa authorization rules rule 200
Value for 'context' (<string>): *
Value for 'path' (<string>): *
Value for 'group' (<string>): *
Value for 'op' (<string>): rw
Value for 'action' [reject,accept]: reject
[ok][2006-06-02 12:31:59]

[edit]
admin@io%
```

This saves the user from typing explicit **set** commands to set each required setting.

Note that it is recommended to disable the autowizard before pasting in a list of commands, in order to avoid prompting. A good practice is to start all such scripts with a line that disables the autowizard:

```
set autowizard false
...
set autowizard true
```

complete-on-space (*true* | *false*)
Controls if command completion should be attempted when <space> is entered. Entering <tab> always results in command completion.

ignore-leading--space (*true* | *false*)
Controls if leading spaces should be ignored or not. This is useful to turn off when pasting commands into the CLI.

history (<*integer*>)
Size of CLI command history.

idle-timeout (<*seconds*>)
Maximum idle time before being logged out. Use 0 (zero) for for infinity.

paginate (*true* | *false*)
Some commands paginate their output, for example. This can be disabled or enabled. It is enabled by default.

screen width (<*integer*>)
Current width of terminal. This is used when paginating output to get proper line count.

screen length (<*integer*>)
Current length of terminal. This is used when paginating output to get proper line count.

service prompt config
Controls whether a prompt should be displayed in configure mode in the C- and I-style CLI. If set to false then no prompt will be displayed. The setting is changed using the commands **no service prompt config** and **service prompt config** in configure mode.

show defaults (*true* | *false*)
Controls if defaults values should be shown when displaying the configuration. The default values are shown as comments after the configured value.

For example:

```
admin@io% show hosts host
host x15 {
    domain tail-f.com;
    defgw 10.1.5.2;
    interfaces {
        interface 1 {
            name eth0;
            ipMask {
                ip 192.68.0.60;   # 10.0.0.0
                mask 255.255.0.0;   # 255.255.0.0
            }
            enabled false;
        }
    }
}
[ok][2006-06-02 12:31:59]

[edit]
admin@io%
```

terminal (*string*)
Terminal type. This setting is used for controlling how line editing is performed. Supported terminals are: dumb, vt100, xterm, linux, and ansi. Other terminals may also work but have no explicit support.

16.19. Commands in J-style

It is possible to get a full XML listing of the commands available in a running ConfD instance by using the confd option --cli-j-dump <file>. The generated file is only intended for documentation purposes and cannot be used as input to confdc.

16.19.1. Operational mode commands

compare startup [**brief**] [<*pathfilter*>]
Compare current configuration to the startup configuration. This command is only available when the system has been configured to have a startup configuration. Differences will be annotated with **-** (removed) and **+** (added). With the **brief** option, only the actual diffs will be shown.

compare file <*file*> [**brief**] [<*pathfilter*>]
Compare current configuration to a configuration stored on file, i.e. previously saved using the **save** command. Differences will be annotated with **-** (removed) and **+** (added). With the **brief** option, only the actual diffs will be shown.

configure (**private** | **exclusive** | **shared**)
Enter configure mode. The default is **private**. The options have slightly different meaning depending on how the system is configured; with a writable running configuration, with a startup configuration, and with a candidate configuration.

private (writable running enabled)
Edit a private copy of the running configuration, no lock is taken.

private (writable running disabled, startup enabled)
: Edit a private copy of the startup configuration, no lock is taken.

exclusive (candidate enabled)
: Lock the running configuration and the candidate configuration and edit the candidate configuration.

exclusive (candidate disabled, startup enabled)
: Lock the running configuration (if enabled) and the startup configuration and edit the startup configuration.

shared (writable running enabled, candidate enabled)
: Edit the candidate configuration without locking it.

Example:

```
admin@io> configure private
              Entering configuration mode "private"
              [ok][2006-06-02 12:31:59]
              [edit]
              admin%
```

file show <*file*>
Display contents of a <*file*>.

Example:

```
admin@io> file show /etc/skel/.bash_profile
              # /etc/skel/.bash_profile

              # This file is sourced by bash for login shells.  The following line
              # runs our .bashrc and is recommended by the bash info pages.
              [[ -f ~/.bashrc ]] &amp;amp;&amp;amp; . ~/.bashrc
              [ok][2006-06-02 12:31:59]
              admin@io>
```

file list <*directory*>
List files in <*directory*>.

Example:

```
admin@io> file list /config
              rollback0
              rollback1
              rollback2
              rollback3
              rollback4
              [ok][2006-06-02 12:31:59]
              admin@io>
```

help <*command*>
Display help text related to <*command*>.

Example:

```
admin@io> help request job
              Help for command: request job
              Job operations
              [ok][2006-06-02 12:31:59]
              admin@io>
```

request <*path*> <*parameters*>

Invokes the action found at *path* using the supplied *parameters*. For example, given the following action specification in a yang file:

```
tailf:action shutdown {
              tailf:actionpoint actions;
              input {
              tailf:constant-leaf flags {
              type uint64 {
              range "1 .. max";
              }
              tailf:constant-value 42;
              }
              leaf timeout {
              type xs:duration;
              default PT60S;
              }
              leaf message {
              type string;
              }
              container options {
              leaf rebootAfterShutdown {
              type boolean;
              default false;
              }
              leaf forceFsckAfterReboot {
              type boolean;
              default false;
              }
              leaf powerOffAfterShutdown {
              type boolean;
              default true;
              }
              }
              }
              }
```

the action can be invoked in the following way

```
admin@io> request shutdown timeout 10s message reboot options { forceFsckAfterReboot tru
```

request system logout user (<*username*> | <*sessionid*>)

Log out a specific user or session from the device. If the user held the **configure exclusive** lock, it will be released.

<username>	Log out a specific user.
<sessionid>	Terminate a specific session.

Example:

```
admin@io> show users
              SID USER  TYPE FROM      PROTO   LOGIN
              *4  admin cli  127.0.0.1 console 13:11:03
              3  oper  cli  127.0.0.1 console 13:11:01
              [ok][2006-06-02 12:31:59]
              admin@io> request system logout user oper
              [ok][2006-06-02 12:31:59]
              admin@io> show users
              SID USER  TYPE FROM      PROTO   LOGIN
```

```
              *4  admin cli  127.0.0.1 console 13:11:03
              [ok][2006-06-02 12:31:59]
              admin@io>
```

request message (**all** | *<user>*) *<message>*
Display a message on the screens of all users who are logged in to the device or on a specific screen.

all
Display the message to all currently logged in users.

<user>
Display the message to a specific user.

Example:

```
admin@io> request message oper "I will reboot system in 5 minutes."
              admin@io>

              oper@io> Message from admin@io at 13:16:41...
              I will reboot system in 5 minutes.
              EOF
```

request job stop *<job id>*
Stop a specific background job. In the default CLI the only command that creates background jobs is **monitor start**.

Example:

```
admin@io> monitor start /var/log/messages
        [ok][2006-06-02 12:31:59]
        admin@io> show jobs
        JOB COMMAND
        3   monitor start /var/log/messages
        [ok][2006-06-02 12:31:59]
        admin@io> request job stop 3
        [ok][2006-06-02 12:31:59]
        admin@io> show jobs
        JOB COMMAND
        [ok][2006-06-02 12:31:59]
        admin@io>
```

set (**complete-on-space** | **ignore-leading-space** | **idle-timeout** | **paginate** | **screen length** | **screen width** | **terminal** |**autowizard** | **show defaults**) *<value>*
Set CLI properties.

Example:

```
admin@io> set autowizard true
        [ok][2006-06-02 12:31:59]
        admin@io>
```

set-path *<path>*
This commands lets you 'cd' into a status part of the tree. Similar to the 'edit' command in configure mode.

show cli
Display CLI properties.

Example:

```
admin@io> show cli
```

```
        cli {
        autowizard true;
        complete-on-space true;
        ignore-leading-space false;
        history 100;
        idle-timeout 1800;
        output {
        file terminal;
        }
        paginate true;
        screen {
        length 82;
        width 80;
        }
        show {
        defaults false;
        }
        terminal xterm;
        }
        [ok][2006-06-02 12:31:59]
        admin@io>
```

show cli history [<*limit*>]

Display CLI command history. By default the last 100 commands are listed. The size of the history list is configured using the history CLI setting. If you specify a history limit, only the last number of commands up to that limit will be shown.

Example:

```
admin@io> show cli history
        06-19 14:34:02 -- ping router
        06-20 14:42:35 -- show configuration
        06-20 14:42:37 -- show users
        06-20 14:42:40 -- show cli history
        [ok][2006-06-20 14:42:40]
        admin@io> show cli history 3
        14:42:37 -- show users
        14:42:40 -- show cli history
        14:42:46 -- show cli history 3
        [ok][2006-06-20 14:42:46]
        admin@io>
```

show all [**details**] [<*pathfilter*>]

Display both configuration and status. By default the whole configuration and the whole status tree is displayed. It is possible to limit what is shown by supplying a pathfilter.

The *pathfilter* may be either a path pointing to a specific instance, or if an instance id is omitted, the part following the omitted instance is treated as a filter.

show configuration [**details**] [<*pathfilter*>]

Display current configuration. By default the whole configuration is displayed. It is possible to limit what is shown by supplying a pathfilter.

The *pathfilter* may be either a path pointing to a specific instance, or if an instance id is omitted, the part following the omitted instance is treated as a filter.

Example:

To show the aaa settings for the admin user, you can do:

```
admin@io> show configuration aaa authentication users user admin
        uid 100;
        gid 10;
        password $1$feedbabe$nGlMYlZpQ0bzenyFOQI3L1;
        ssh_keydir /var/confd/homes/admin/.ssh;
        homedir /var/confd/homes/admin;
        [ok][2006-07-31 17:16:01]
        admin@io>
```

To show all users that have group id 10, you would omit the user id, and instead specify gid 10.

```
admin@io> show configuration aaa authentication users user gid 10
        user admin {
        uid 100;
        gid 10;
        password $1$feedbabe$nGlMYlZpQ0bzenyFOQI3L1;
        ssh_keydir /var/confd/homes/admin/.ssh;
        homedir /var/confd/homes/admin;
        }
        user oper {
        uid 100;
        gid 10;
        password $1$feedbabe$i2glnaB.iUj2VXh/zlq.o/;
        ssh_keydir /var/confd/homes/oper/.ssh;
        homedir /var/confd/homes/oper;
        }
        [ok][2006-07-31 17:16:56]
        admin@io>
```

show parser dump <*command prefix*>
Shows all possible commands starting with *command prefix*.

show status <*pathfilter*>
Display current values of read-only nodes in the configuration.

show table <*path*>
This command shows the configuration as a table provided that *path* leads to a list element.

Example:

```
admin@io 09:42:08> show table aaa authentication groups group
        NAME   GID  USERS
        --------------------------
        admin       admin private
        oper        oper public
        [ok][2007-09-17 09:44:12]

        [edit]
        admin@io 09:44:12>
```

show users
Display currently logged on users. The current session, i.e. the session running the show status command, is marked with an asterisk.

Example:

```
admin@io> show users
        SID USER  TYPE FROM      PROTO   LOGIN
        *4  admin cli  127.0.0.1 console 13:11:03
        3  oper  cli  127.0.0.1 console 13:11:01
```

```
        [ok][2006-06-02 12:31:59]
        admin@io>
```

show jobs
Display currently running background jobs.

Example:

```
admin@io> show jobs
        JOB COMMAND
        3   monitor start /var/log/messages
```

show log <*file*>
Display contents of a log file.

Example:

```
admin@io> show log messages
```

source <*file*>
Execute commands from <file> as if they had been entered by the user. The autowizard is disabled when executing commands from the file.

commit (**abort** | **confirm**) [**persist-id** <*id*>]
Abort or confirm a pending confirming commit. A pending confirming commit will also be aborted if the CLI session is terminated without doing **commit confirm**. The default is confirm.

If the confirming commit has been initiated with the persist option then user needs to supply the same token as a **persist-id** for the **commit** to have effect.

Example:

```
admin@io> commit abort
```

describe <*command*>
Display detailed information about a command.

Example:

```
admin@io> describe ping
Common
    Source : clispec
    File   : commands-c.cli

Callback [os command]
    OS command  : ping
    Run as user : confd

Help
    Verify IP (ICMP) connectivity to a host.

Info
    Ping a host
```

16.19.2. Configure mode commands

activate <*statement*>
Activate a statement that has previously been deactivated.

Only available in when the system has been configured with support for inactive.

deactivate <*statement*>
Deactivate a statement in the configuration.

Only available in when the system has been configured with support for inactive.

annotate <*statement*> <*text*>
Associate an annotation with a given configuration statement. To remove an annotation leave the text empty.

Only available in when the system has been configured with attributes enabled.

commit (**check** | **and-quit** | **confirmed** [<*timeout*>] [**persist** <*token*>] | **to-startup**) [**comment** <*text*>] [**label** <*text*>] [**persist-id** <*id*>]
Commit current configuration to running.

check
Validate current configuration.

and-quit
Commit to running and quit configure mode.

to-startup
Commit current configuration to running and startup configuration. Only available if the system is configured to have a startup configuration.

confirmed
Commits the current configuration to running with a timeout. If no **commit confirm** command has been issued before the timeout expires, then the configuration will be reverted to the configuration that was active before the **commit confirmed** command was issued. If no timeout is given then the confirming commit will have a timeout of 10 minutes. The configuration session will be terminated after this command since no further editing is possible. Only available in **configure exclusive** and **configure shared** mode when the system has been configured with a candidate.

The confirming commit will be rolled back if the CLI session is terminated before confirming the commit, unless the **persist** argument is given. If the **persist** command is given then the CLI session can be terminated and a later session may confirm the pending commit by supplying the persist token as an argument to the **commit** command using the **persist-id** argument.

comment <*text*>
Associate a comment with the commit. The comment can later be seen when examining rollback files.

label <*text*>
Associate a label with the commit. The label can later be seen when examining rollback files.

persist-id <*id*>
If a prior confirming commit operation has been performed with the **persist** argument, then to modify the ongoing confirming commit process the **persist-id** argument needs to be supplied with the same persist token. This makes it possible to, for example, abort an onging *persist* commit, or extend the timeout.

validate
Validates current configuration. This is the same operation as **commit check**.

insert <*path*>
Inserts a new element. If the element already exists and has the indexedView option set in the data model, then the old element will be renamed to element+1 and the new element inserted in its place.

insert <*path*> [**first** | **last**| **before** *key*| **after** *key*]
Insert a new element into an ordered list. The element can be added first, last (default), before or after another element.

move <*path*>[**first**|**last**|**before***key*|**after***key*]
Move an existing element to a new position in an ordered list. The element can be moved first, last (default), before or after another element.

rename <*instance path*> <*new id*>
Rename an instance.

delete <*path*>
Delete a data element.

edit <*path*>
Edit a sub-element. Missing elements in *path* will be created.

exit (**level** | **configuration-mode**)

level	Exit from this level. If performed on the top level, will exit configure mode. This is the default if no option is given.
configuration-mode	Exit from configuration mode regardless of which edit level.

help <*command*>
Shows help text for command.

hide <*hide-group*>
Re-hides the elements and actions belonging to the hide groups. No password is required for hiding. Note that this command is hidden and not shown during command completion.

unhide <*hide-group*>
Unhides all elements and actions belonging to the hide-group. You may be required to enter a password. Note that this command is hidden and not shown during command completion

load (**merge** | **replace** | **override**) (<*file*> | **terminal**)
Load configuration from file or terminal.

merge
Merge content of file/terminal with current configuration.

replace
Replace the content of file/terminal for the corresponding parts of the current configuration. This is different from `override` in the that only the parts that occur in the file/terminal are replace, the rest of the configuration is left as is. In the case of `override` the entire configuration is deleted (with the exception of hidden data) before loading the new configuration from the file/terminal.

There currently exists a discrepancy in behavior between different CLI:s and file formats. List nodes will be replaced for the following combinations:

- Juniper CLI, XML and curly bracket format.
- Cisco CLI, XML format.

List nodes will be merged for the following combination:

- Cisco CLI, curly bracket format.

override The current configuration is deleted and a new configuration is loaded from file/terminal. Hidden data is not affected.

The configuration file may contain **replace:** and **delete:** directives. For example if you have the configuration

```
system {
    parent-mo {
        child-mo 1 {
            attr 10;
        }
        child-mo 2 {
            attr 5;
        }
    }
}
```

and want to delete **child-mo 2**, you can create a configuration file containing either (using **replace:**)

```
system {
    replace:
    parent-mo {
        child-mo 1 {
            attr 2;
        }
    }
}
```

or (using **delete:**)

```
system {
    parent-mo {
        delete:
        child-mo 2 {
            attr 5;
        }
    }
}
```

save <*file*> [*xml*] [<*pathfilter*>]

Save the whole or parts of the current configuration to file. By the default the configuration is saved in curly bracket format. If the *xml* argument is given then the configuration is saved in XML format.

rollback [<*number*>]

Return the configuration to a previously committed configuration. The system stores a limited number of old configurations. The number of old configurations to store is configured in the `confd.conf` file. If more than the configured number of configurations are stored, then the oldest configuration is removed before creating a new one.

The most recently committed configuration (the running configuration) is number 0, the next most recent 1, etc.

The files are called rollback0 - rollback*X*, where *X* is the maximum number of saved committed configurations.

Example:

```
admin@io% rollback 1
[ok][2006-06-02 12:31:59]
admin@io%
```

Note that this command is only available if rollback has been enabled in confd.conf.

run <*command*>
Run command in operational mode.

set <*path*> [<*value*>]
Set a parameter. If a new identifier is created and **autowizard** is enabled, then the CLI will prompt the user for all mandatory sub-elements of that identifier. It will also prompt for mandatory action parameters.

If no <*value*> is provided, then the CLI will prompt the user for the value. No echo of the entered value will occur if <*path*> is an encrypted value, i.e. of the type *tailf:md5-digest-string*, *tailf:des3-cbc-encrypted-string*, or *tailf:aes-cfb-128-encrypted-string* as described in confd_types(3).

show [**details**] [<*pathfilter*>]
Show current configuration. The show command can be limited to a part of the configuration by providing a <*pathfilter*>.

show parser dump <*command prefix*>
Shows all possible commands starting with *command prefix*.

compare running [**brief**] [<*pathfilter*>]
Compare current configuration to the running configuration. Differences will be annotated with **-** (removed) and **+** (added). With the *brief* option, only the actual diffs will be shown.

compare startup [**brief**] [<*pathfilter*>]
Compare current configuration to the startup configuration. This command is only available when the system has been configured to have a startup configuration. With the **brief** option, only the actual diffs will be shown.

compare file <*file*> [**brief**] [<*pathfilter*>]
Compare current configuration to a configuration stored on file, i.e. previously saved using the **save** command. With the **brief** option, only the actual diffs will be shown.

tag add <*statement*> <*tag*>
Add a tag to a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag del <*statement*> <*tag*>
Remove a tag from a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag clear <*statement*>
Remove all tags from a configuration statement.

Only available in when the system has been configured with attributes enabled.

top [*command*]
Exit to top level of configuration, or execute a command at the top level of the configuration.

up [*command*]
Exit one level of configuration, or execute a command at one level up.

revert
Copy running configuration into current configuration.

status
Display users currently editing the configuration.

describe [<*path*> | <*command*>]
Display detailed information about a command. This information may for example consist of the source of the command (YANG, clispec or built-in), the corresponding path in the YANG file (in case of an auto-rendered command) and information regarding what callpoints, actionpoints and validation points that may be tied to the command. Due to the verbose information that may be displayed, it may be desirable to restrict the usage of this command by including it in an appropriate set of authorization rules or by the means of any other authorization functionality.

Example:

```
admin@io% describe dhcp
Common
    Source          : YANG
    Module          : dhcpd
    Namespace       : http://tail-f.com/ns/example/dhcpd
    Path            : /dhcp
    Node            : container
    Exported agents : all
    Checksum        : 3c893927631cceee3700c23bb38cd050
```

16.20. Commands in C/I-style

It is possible to get a full XML listing of the commands available in a running ConfD instance by using the confd option --cli-c-dump <file>, for C-style and --cli-i-dump for I-style. The generated file is only intended for documentation purposes and cannot be used as input to confdc.

16.20.1. Operational mode commands

The IOS CLI does not have any of the commands associated with transactions, i.e. commit, abort, show configuration.

The IOS CLI has commands for entering and leaving privileged mode, settings passwords and secrets and assigning privilege levels to commands in EXEC mode.

The privilege information for the IOS mode is stored in the AAA namespace under the aaa/ios element. Note that all EXEC commands are available at level 15 without needing to specify them in the aaa-configuration.

It is possible to assign custom prompt to the different levels in EXEC mode. The default is for level 0 to have the "\h> " prompt and for all other levels to have the "\h# " prompt.

compare startup [**brief**] [<*pathfilter*>]
Compare current configuration to the startup configuration. This command is only available when the system has been configured to have a startup configuration. Differences will be annotated with **-** (removed) and **+** (added). With the **brief** option, only the actual diffs will be shown.

compare file <*file*> [**brief**] [<*pathfilter*>]
Compare current configuration to a configuration stored on file, i.e. previously saved using the **save** command. Differences will be annotated with **-** (removed) and **+** (added). With the **brief** option, only the actual diffs will be shown.

config (**terminal** | **shared** | **exclusive**)

Enter configure mode. The default is **terminal**. The options have slightly different meaning depending on how the system is configured; with a writable running configuration, with a startup configuration, and with a candidate configuration.

terminal (writable running enabled)

Edit a private copy of the running configuration, no lock is taken.

terminal (writable running disabled, startup enabled)

Edit a private copy of the startup configuration, no lock is taken.

exclusive (candidate enabled)

Lock the running configuration and the candidate configuration and edit the candidate configuration.

exclusive (candidate disabled, startup enabled)

Lock the running configuration (if enabled) and the startup configuration and edit the startup configuration.

shared (writable running enabled, candidate enabled)

Edit the candidate configuration without locking it.

Example:

```
io# config terminal
Entering configuration mode terminal
io(config)#
```

enable (**<level>**)

Only available in IOS (i) mode. Enables privileged EXEC commands. The default level is 15. The CLI will prompt for a password if a password has been assigned to the level.

Example:

```
io> enable
io#
```

disable (**<level>**)

Only available in IOS (i) mode. Downgrade to a lower privilege level.

Example:

```
io# disable 4
io#
```

file show <*file*>

Display contents of a <*file*>.

Example:

```
io# file show /etc/skel/.bash_profile
# /etc/skel/.bash_profile

# This file is sourced by bash for login shells.  The following line
# runs your .bashrc and is recommended by the bash info pages.
[[ -f ~/.bashrc ]] &amp;&amp; . ~/.bashrc
io#
```

file list <*directory*>
List files in <*directory*>.

Example:

```
io# file list /config
rollback0
rollback1
rollback2
rollback3
rollback4
io#
```

help <*command*>
Display help text related to <*command*>.

Example:

```
io# help config
Help for command: config
    Manipulate software configuration information
io#
```

id
Show user id information; uid, gid, and groups

Example:

```
io# id
user = joe(1000), gid=100, groups=wheel, gids=10,100
io#
```

message (**all** | <*user*>) <*message*>
Display a message on the screens of all users who are logged in to the device or on a specific screen.

all
Display the message to all currently logged in users.

<*user*>
Display the message to a specific user.

Example:

```
io# message all "I will reboot the system in 5 minutes."
Message from joe@io at 13:26:49...
I will reboot the system in 5 minutes.
EOF
io#
```

job stop <*job id*>
Stop a specific background job. In the default C-style CLI there are no commands that create background jobs. Custom commands can be created that do this.

Example:

```
io# show jobs
JOB COMMAND
1   monitor start /tmp/saved
io# job stop 1
io# show jobs
```

```
JOB COMMAND
io#
```

show cli
Display CLI properties.

Example:

```
io# show cli
autowizard           true
complete-on-space    true
ignore-leading-space false
history              50
idle-timeout         1800
output-file          terminal
paginate             true
screen-length        82
screen-width         80
show-defaults        false
terminal             xterm
io#
```

history [<*limit*>]
Display CLI command history. By default the last 100 commands are listed. The size of the history list is configured using the CLI history setting. If you specify a history limit, only the last number of commands up to that limit will be shown.

Example:

```
io# history
13:28:46 -- show jobs
13:28:53 -- job stop 1
13:29:05 -- show jobs
13:29:51 -- show
13:29:53 -- show cli
13:30:31 -- history
io#
```

show configuration commit list <*id*>
List rollback files (C-style only). Note that this command is only available if rollback has been enabled in confd.conf.

show notification stream <*event stream name*> [**last** <*number of events*>] [**from** <*dateTime (ccyy-mm-dd|hh:mm:ss|ccyy-mm-ddThh:mm:ss)*>] [**to** <*dateTime (ccyy-mm-dd|hh:mm:ss|ccyy-mm-ddThh:mm:ss)*>]
Display the last notifications in a selected stream. The stream must use the builtin store and have replay enabled. It is possible to limit the output by specifying the maximum number of events and/or a time range.

show parser dump <*command prefix*>
Shows all possible commands starting with *command prefix*.

show running-config [**details | all**] [<*pathfilter*>]
Display current configuration. By default the whole configuration is displayed. It is possible to limit what is shown by supplying a pathfilter. The pathfilter may be either a path pointing to a specific instance, or if an instance id is omitted, the part following the omitted instance is treated as a filter.

Example:

To show the aaa settings for the admin user, you can do:

```
io# show running-config aaa authentication users user admin
aaa authentication users user admin
uid        1000
gid        100
password   $1$fB$0w68PmacQ4VmE3/M3nK3Ug==
ssh_keydir /var/confd/homes/admin/.ssh
homedir    /var/confd/homes/admin
!
io#
```

To show all users that have group id 10, you would omit the user id, and instead specify gid 10.

```
io# show running-config aaa authentication users user gid 100
aaa authentication users user admin
uid        1000
gid        100
password   $1$fB$0w68PmacQ4VmE3/M3nK3Ug==
ssh_keydir /var/confd/homes/admin/.ssh
homedir    /var/confd/homes/admin
!
aaa authentication users user oper
uid        1000
gid        100
password   $1$S6$brGZW9wSDifHoU7Rf5KSHA==
ssh_keydir /var/confd/homes/oper/.ssh
homedir    /var/confd/homes/oper
!
```

Per default only elements that have been explicitly set to a value are shown. This makes it easier to handle large configurations. However, it is possible to force the show command to display all elements. This is done using the 'details' or 'all' options.

show startup-config [**details** | **all**] [<*pathfilter*>]
: Display the startup configuration. This command is only available if ConfD has been configured with a startup configuration. By default the whole configuration is displayed. It is possible to limit what is shown by supplying a pathfilter. The pathfilter may be either a path pointing to a specific instance, or if an instance id is omitted, the part following the omitted instance is treated as a filter.

  Per default only elements that have been explicitly set to a value are shown. This makes it easier to handle large configurations. However, it is possible to force the show command to display all elements. This is done using the 'details' or 'all' options.

write terminal [<*pathfilter*>]
: Display current configuration.

copy running-config startup-config
: Copy running configuration to startup configuration. Only available when the system has been configured to have a startup database.

write memory
: Copy running configuration to startup configuration. Only available when the system has been configured to have a startup database.

source <*file*>
: Execute commands from <file> as if they had been entered by the user. The autowizard is disabled when executing commands from the file.

show <*pathfilter*>
Display current values of read-only parameters. If a list element is encountered then the command attempts to arrange the output as a table.

who
Display currently logged on users. The current session, i.e. the session running the show status command, is marked with an asterisk.

Example:

```
io# who
Session User Context From      Proto   Date
*7      joe  cli     127.0.0.1 console 13:19:05
io#
```

logout (<*username*> | <*sessionid*>)
Terminates the current session.

logout (<*username*> | <*sessionid*>)
Log out a specific user or session from the device. If the user held the **configure exclusive** lock, it will be released.

<username> Log out a specific user.

<sessionid> Terminate a specific session.

Example:

```
io# who
Session User  Context From      Proto   Date
*5      admin cli     127.0.0.1 console 10:25:46
4      jb    cli     127.0.0.1 console 10:25:37
io# logout jb
io# who
Session User  Context From      Proto   Date
*5      admin cli     127.0.0.1 console 10:25:46
io#
```

show jobs
Display currently running background jobs.

Example:

```
io# show jobs
JOB COMMAND
2   monitor start /tmp/saved
io#
```

show log <*file*>
Display contents of a log file.

Example:

```
io# show log messages
```

commit (**abort** | **confirm**) [**persist-id** <*id*>]
Abort or confirm a pending confirming commit. A pending confirming commit will also be aborted if the CLI session is terminated without doing **commit confirm**. The default is confirm.

If the confirming commit was initiated with a *persist* argument then the same token needs to be supplied using the **persist-id** argument to this command.

Example:

```
io# commit abort
```

timestamp (**enable** | **disable**)
Display a timestamp after a command has been executed. The timestamp is displayed in the timezone UTC+-00:00 by default. A UTC offset may be configured in confd.conf.

Example:

```
io# timestamp enable
io# config
Tue Mar  12 11:31:03.698 UTC
Entering configuration mode terminal
io(config)#
```

describe <*command*>
Display detailed information about a command.

Example:

```
# describe ping
Common
    Source : clispec
    File   : commands-c.cli

Callback [os command]
    OS command  : ping
    Run as user : confd

Help
    Verify IP (ICMP) connectivity to a host.

Info
    Ping a host
```

16.20.2. Configure mode commands

alias (<*alias-name*> <*alias-expansion*>)
Defines the alias *alias-name*. It will be expanded to *alias-expansion*.

It is possible to define parametrised aliases, i.e. aliases that accepts parameters. The parameters are then expanded when the alias is applied.

The alias can be used anywhere on the command line. After a command with an alias has been entered, the expanded command line is displayed so that you can verify the alias value.

For example:

```
io(config)# alias foo(a,b) "show $(a) ; show $(b)"
io(config)# alias myUser c87923
io(config)# commit
io(config)# foo(history,configuration)
io(config)# show history ; show configuration
```

```
...
io(config)# aaa authentication users user myUser
io(config)# aaa authentication users user c87923
```

The aliases are stored persistently in cdb in the aaa namespace. Note that the clispec file needs to contain the following three entries in the modifications section:

```
<dropElem src="alias expansion"/>
<multiValue src="alias expansion"/>
<suppressMode src="alias "/>
```

activate <*statement*>
Activate a statement in the configuration that has previously been deactivated.

Only available in when the system has been configured with support for inactive.

deactivate <*statement*>
Deactivate a statement in the configuration.

Only available in when the system has been configured with support for inactive.

annotate <*statement*> <*text*>
Associate an annotation with a given configuration statement. To remove an annotation leave the text empty.

Only available in when the system has been configured with attributes enabled.

tag add <*statement*> <*tag*>
Add a tag to a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag del <*statement*> <*tag*>
Remove a tag from a configuration statement.

Only available in when the system has been configured with attributes enabled.

tag clear <*statement*>
Remove all tags from a configuration statement.

Only available in when the system has been configured with attributes enabled.

enable (**secret | password**) **level <level>** (**0 | 7**) **<password>**
Only available in IOS (i) mode. Configures a password for a specific EXEC level. If both a password and a secret is configured, the secret is used.

secret | password	Specifies how the password is to be encrypted
<*level*>	The EXEC level to password protect
0 | 7	Use 0 to indicate that the password given at the end of the command is in plain text, and 7 for an already encrypted password.
<*password*>	The actual password

Example:

```
io(config)# enable secret level 3 0 bluebox
```

```
io(config)#
```

privilege <mode> level <level> <command>
Only available in IOS (i) mode. Configures for which level a command should be available.

<mode> In which mode is the command.

<*level*> In which privilege level should the command be available

<*command*> Command string.

Example:

```
io(config)# privilege exec level 4 show
io(config)#
```

hide <*hide-group*>
Re-hides the elements and actions belonging to the hide groups. No password is required for hiding. Note that this command is hidden and not shown during command completion.

unhide <*hide-group*>
Unhides all elements and actions belonging to the hide-group. You may be required to enter a password. Note that this command is hidden and not shown during command completion

commit (**check** | **and-quit** | **confirmed** [<*timeout*>] [**persist** <*token*>] **to-startup**) [**comment** <*text*>] [**label** <*text*>] [**persist-id** <*id*>]
Commit current configuration to running.

check
Validate current configuration.

and-quit
Commit to running and quit configure mode.

to-startup
Commit current configuration to running and startup configuration. Only available if the system is configured to have a startup configuration.

confirmed
Commits the current configuration to running with a timeout. If no **commit confirm** command has been issued before the timeout expires, then the configuration will be reverted to the configuration that was active before the **commit confirmed** command was issued. If no timeout is given then the confirming commit will have a timeout of 10 minutes. The configuration session will be terminated after this command since no further editing is possible. Only available in **configure exclusive** and **configure shared** mode when the system has been configured with a candidate.

The confirming commit will be rolled back if the CLI session is terminated before confirming the commit, unless the **persist** argument is given. If the **persist** command is given then the CLI session can be terminated and a later session may confirm the pending commit by supplying the persist token as an argument to the **commit** command using the **persist-id** argument.

comment <*text*>
Associate a comment with the commit. The comment can later be seen when examining rollback files.

label <*text*>
Associate a label with the commit. The label can later be seen when examining rollback files.

persist-id <*id*>
If a prior confirming commit operation has been performed with the **persist** argument, then to modify the ongoing confirming commit process the **persist-id** argument needs to be supplied with the same persist token. This makes it possible to, for example, abort an onging *persist* commit, or extend the timeout.

validate
Validates current configuration. This is the same operation as **commit check**.

insert <*path*>
Inserts a new element. If the element already exists and has the indexedView option set in the data model, then the old element will be renamed to element+1 and the new element inserted in its place.

insert <*path*>[**first**|**last**|**before***key*|**after***key*]
Insert a new element into an ordered list. The element can be added first, last (default), before or after another element.

move <*path*>[**first**|**last**|**before***key*|**after***key*]
Move an existing element to a new position in an ordered list. The element can be moved first, last (default), before or after another element.

rename <*instance path*> <*new id*>
Rename an instance.

no <*path*>
Delete or unsets a data element.

exit (**level** | **configuration-mode**)

level	Exit from current mode. If performed on the top level, will exit configure mode. This is the default if no option is given.
configuration-mode	Exit from configuration mode regardless of mode.

help <*command*>
Shows help text for command.

load (<*file*>)
Load configuration from file.

pwd
Display current submode path.

save <*file*> [*xml*] [<*pathfilter*>]
Save the whole or parts of the current configuration to file.

rollback configuration [<*number*>]
Return the configuration to a previously committed configuration. The system stores a limited number of old configurations. The number of old configurations to store is configured in the confd.conf file. If more than the configured number of configurations are stored, then the oldest configuration is removed before creating a new one.

The most recently committed configuration (the running configuration) is number 0, the next most recent 1, etc.

The files are called rollback0 - rollback*X*, where *X* is the maximum number of saved committed configurations.

Example:

```
io(config)# rollback configuration 1
io#
```

Note that this command is only available if rollback has been enabled in confd.conf.

do <*command*>
Run command in operational mode.

show configuration [<*pathfilter*>]
Show current configuration buffer. The show command can be limited to a part of the configuration by providing a <*pathfilter*>.

show configuration diff [<*pathfilter*>]
Show configuration changes in diff style, ie new lines prefixed with a plus (+) sign, and removed lines prefixed with a minus (-) sign.. The show command can be limited <*pathfilter*>.

show configuration commit changes <*id*>
Show changes that were committed for a given commit *id*.

show configuration commit list <*id*>
List rollback files (C-style only). Note that this command is only available if rollback has been enabled in confd.conf.

show configuration rollback changes <*nr*>
Show changes for rolling back to rollback file *nr*.

show full-configuration [**details**] [<*pathfilter*>]
Show current configuration. The show command can be limited to a part of the configuration by providing a <*pathfilter*>.

show parser dump <*command prefix*>
Shows all possible commands starting with *command prefix*.

revert
Copy running configuration into current configuration.

clear
Remove all configuration changes.

describe [<*path*> | <*command*>]
Display detailed information about a command. This information may for example consist of the source of the command (YANG, clispec or built-in), the corresponding path in the YANG file (in case of an auto-rendered command) and information regarding what callpoints, actionpoints and validation points that may be tied to the command. Due to the verbose information that may be displayed, it may be desirable to restrict the usage of this command by including it in an appropriate set of authorization rules or by the means of any other authorization functionality.

Example:

```
(config)# describe dhcp
Common
    Source          : YANG
    Module          : dhcpd
    Namespace       : http://tail-f.com/ns/example/dhcpd
    Path            : /dhcp
    Node            : container
```

```
   Exported agents : all
   Checksum        : 3c893927631cceee3700c23bb38cd050
```

16.21. Customizing the CLI

16.21.1. Modifying builtin commands

There are a number of built-in commands in each CLI style. These can be modified in a number of ways. It is possible to remove them, rename them, hide them, change their help and info texts, add a command timeout and to add confirmation prompts. This is done using the <*modifications*> section of the *clispec* file. For example:

```
<modifications>
  <confirmText src="configure">
    Are your really sure you know what you are doing?
  </confirmText>
  <delete src="request system logout"/>
  <delete src="show log"/>
  <help src="configure private">
    Enter private configuration mode.
  </help>
  <info src="configure">Hi there info configure!</info>
  <help src="configure">Hi there help configure!</help>
  <timeout src="show configuration">
  10
  </timeout> -->
</modifications>
```

See the *clispec.5* man page for a detailed description of each modifications option.

16.21.2. Adding new commands

New commands can be added in two different ways, either as an action in the *YANG* file, or as a command in the *clispec* file. The advantage of using an action is that the command will be available through all north-bound interfaces. However, a clispec command may give you better control over your input parameters.

It is also possible to use a dedicated data model for the CLI, i.e. a YANG file that is only visible (exported) in the CLI. This is useful if you want to add custom modes in the C- and I-style CLIs.

16.21.3. Suppressing automatically generated modes

The C- and I- style CLIs will automatically create new modes for all list elements in the configuration, i.e. all elements that have *maxOccurs* larger than 1.

It is possible to suppress this behavior by adding a *suppressMode* direction in the modifications section of the clispec file.

For example:

```
<modifications>
  <suppressMode src="config hosts host"/>
</modifications>
```

The *src* attribute is the clispec path as it appears in the CLI. It should be the path to a list element.

16.21.4. Creating new configuration modes

Similarly to suppressing an automatically generated mode it is sometimes desirable to create modes at points in the configuration where there isn't a list element. This can be done trough a declaration in the *modifications* section of the clispec file.

For example:

```
<modifications>
  <addMode src="config hosts"/>
</modifications>
```

The *src* attribute should be a path to a static/internal element, i.e. an element with *minOccurs="1" maxOccurs="1"*.

16.21.5. Custom mode names

ConfD automatically assigns a mode name based on the element name and the key elements of the container. There are two styles, *full* and *short*, where the *short* style only contains the last element of the path and the *full* all components of the path. Which style to use is configured in the confd.conf file.

The automatic mode name can be overridden either by a fixed mode name or by a dynamically generated mode name. It is done through a *modeName* declaration in the *modifications* section of the clispec file.

For example:

```
<modifications>
  <modeName src="config hosts">
    <fixed>hosts</fixed>
  </modeName>
  <modeName src="config hosts host interfaces interface">
    <capi><cmdpoint>custommode</cmdpoint></capi>
  </modeName>
</modifications>
```

ConfD will invoke the cmdpoint/action *custommode* the first time the mode is entered for a given instance. It will then cache the mode name. The callback will receive the path to the instance as argv/argc arguments and is expected to reply by calling the function confd_action_reply_command().

An example from `actions.c` in the `cli/c_cli` example:

```
static int do_modename(struct confd_user_info *uinfo,
                       char *path, int argc, char **argv)
{
    int i;
    int sock;
    char *mode = "config-priv";

    printf("do_modename called\n");

    printf("path: %s\n", path);
    for (i = 0; i < argc; i++) {
        printf("param %2d: %s\n", i, argv[i]);
    }

    if (confd_action_reply_command(uinfo, &amp;mode, 1) < 0)
        confd_fatal("Failed to reply to confd\n");
```

```
    return CONFD_OK;
}
```

16.21.6. Adding custom show output

It is possible to override the default output from the auto-rendered show commands but not for the J-style 'show configuration', 'show status', 'show all', and 'show table'. It can be done in two different ways. Either using a *show* element in the clispec file or as a regular CLI command.

For example:

```
<show path="aaa authentication users user">
  <callback>
    <exec>
      <osCommand>./aaa_auth.sh</osCommand>
    </exec>
  </callback>
</show>
```

The aaa_auth.sh executable will be invoked whenever the user enters "show aaa authentication users user" or a path prefix. The command will receive the indent depth and path as arguments. The callback can be either a C-function (capi), or an executable, and it will be invoked as an ordinary CLI command.

It is also possible to create an ordinary CLI command and mount it on the same path as the show command target.

For example:

```
<cmd name="group" mount="show aaa authentication groups">
  <help></help>
  <info></info>
  <callback>
    <exec>
      <osCommand>./aaa_group.sh</osCommand>
    </exec>
  </callback>
</cmd>
```

The advantage with this approach is that the command may take additional parameters, for example 'details' or 'statistics'.

16.21.7. Command parameters

A custom CLI command can be defined to have a set of parameters. These parameters can be arranged as a mix of:

- a straight list of command options
- a parameter tree
- a choice list of parameters, where a minimum and maximum number of required parameters can be specified

Suppose you want a command that accepts a number of options in an arbitrary order. For example:

```
show alarm [type <type>] [severity <minor|major>] [acknowledged] [active]
```

The parameters can be entered in arbitrary order and they are all optional. This can be achieved with a choice params list with min and max set.

```
<params mode="choice" min="0" max="4">
  <param>
    <info/><help/>
    <name>type</name>
    <prefix>--type </prefix>
  </param>
  <param>
    <info/><help/>
    <name>severity</name>
    <type><enums>major minor</enums></type>
    <prefix>--severity </prefix>
  </param>
  <param>
    <info/><help/>
    <type><enums>acknowledged</enums></type>
    <prefix>--acknowledged</prefix>
  </param>
  <param>
    <info/><help/>
    <type><enums>active</enums></type>
    <prefix>--active</prefix>
  </param>
</params>
```

Suppose instead that you want the following command:

```
show alarm [type <type> severity <minor|major>] [acknowledged] [active]
```

That is, if you enter *type* you must also enter *severity*. You can specify this with a parameter tree where *severity* is a child to the *type* parameter.

```
<params mode="choice" min="0" max="3">
  <param>
    <info/><help/>
    <name>type</name>
    <prefix>--type </prefix>
    <params>
      <param>
        <info/><help/>
        <name>severity</name>
        <type><enums>major minor</enums></type>
        <prefix>--severity </prefix>
      </param>
    </params>
  </param>
  <param>
    <info/><help/>
    <type><enums>acknowledged</enums></type>
    <prefix>--acknowledged</prefix>
  </param>
  <param>
    <info/><help/>
    <type><enums>active</enums></type>
    <prefix>--active</prefix>
  </param>
</params>
```

You can have nested params lists where some are of choice mode and others are straight lists. However, in a choice list there cannot be two possible paths given a token, i.e., you cannot have two items that have the same name, or one named item and one that accepts a generic value. All paths must be deterministic.

For example, you cannot have the following command parameters:

```
<params mode="choice" min="0" max="2">
  <param>
    <info/><help/>
    <prefix>--type </prefix>
  </param>
  <param>
    <info/><help/>
    <type><enums>major minor</enums></type>
    <prefix>--severity </prefix>
  </param>
</params>
```

The reason is that if you enter the parameter major, then the CLI parser cannot determine if it is intended as the first parameter or the second. The first parameter accepts any input.

16.21.8. Hiding parts of the configuration

It is possible to hide parts of the configuration using the hidden attribute in the yang files, and the hideGroup attribute in the clispec file. All elements with the same hidden attribute belong to the same 'hide group'

For example (yang):

```
leaf resetuser {
  tailf:hidden debug;
  type boolean;
  default false;
}
```

For example (clispec):

```
<cmd name="test">
  <help/>
  <info/>
  <callback>
    <exec>
      <osCommand>
        test.sh
      </osCommand>
    </exec>
  </callback>
  <options>
    <hideGroup>debug</hideGroup>
  </options>
</cmd>
```

In the yang example the *resetuser* leaf belongs to the *debug* hide group. All hidden yang elements must have default values or be optional since they cannot be configured by the user unless they have been unhidden.

Elements hidden in this way will be hidden to users when using the Web UI or the CLI, but can optionally be made visible, i.e. unhidden, through a hidden CLI command. An entry in the confd.conf file is needed to make this possible.

For example:

```
<hideGroup>
  <name>debug</name>
  <password>verysecret</password>
</hideGroup>
```

The debug hide group can be 'unhidden' in the CLI provided that the user knows the name of the hide group and the password. It is possible to leave out the password parameter in which case the CLI will not prompt for one. The *unhide* and *hide* commands are used to interactively hide and unhide hide groups in the CLI.

It is also possible to define a C-callback that is used to authenticate the user. This is useful when you want short lived 'unhide' password, or user-specific unhide password.

Note that CLI commands and actions can also be hidden in the same way, but not individual action parameters. Also, hidden elements are only hidden from the CLI and the Web UI unless the special hide group *full* is used, in which case it is hidden from all northbound interfaces as well as the rollback file.

The special hide groups *full* is useful when the data is only used by the application and should not be part of the actual configuration of the device.

16.21.9. EXEC commands

The I-style style has an additional concept of privilege levels. Only a subset of commands are initially available when the user logs on to the box. It is then possible to enable a higher privilege level using the *enable* command. Entering a new privilege level may require a password and the prompt may change as a result of entering the level.

The AAA data model contains a section for controlling which commands are available in the different levels, which prompt to use for each level and if a password or secret is configured.

The configuration mode commands *enable* and *privilege* are used for dynamically modifying this configuration. The initial configuration should be supplied in the aaa_init.xml file.

16.21.10. File access

The default behavior is to enforce Unix style access restrictions. That is, the users *uid*, *gid*, and *gids* are used to control what the user has read and write access to.

However, it is also possible to jail a CLI user to its home directory (or the directory where confd_cli is started). This is controlled using the confd.conf parameter */confdConfig/cli/restrictedFilaAccess*. If this is set to *true*, then the user only has access to the home directory, or if a directory is specified in a cli command parameter (params/param/type/directory{wd} or params/param/type/file{wd}) to that directory.

16.21.11. Help texts

Help and information texts are specified in a number of places. In the yang files the *tailf:info* element is used to specify a descriptive text that is shown when the user enters ? in the CLI. The first sentence of the *description* text is used when showing one-line descriptions in the CLI, e.g.:

In the YANG file:

```
list ifs {
  tailf:info "Configure interfaces";
  ...
}
list hosts {
  tailf:info "Configure hosts";
  ...
}
list routing-protocol {
  tailf:info "Routing protocols";
  ...
```

```
}
```

In the CLI:

```
io(config)# config ?
Possible completions:
  hosts                    Configure hosts
  ifs                      Configure interfaces
  routing-protocol         Routing protocols
io(config)# config
```

It is also possible to specify help texts for simpleTypes in the YANG files. For example:

```
typedef service-type {
    type enumeration {
        enum http { tailf:info "Web server"; }
        enum smtp { tailf:info "Mail server"; }
    }
}

typedef service-name {
    tailf:info "Name of service";
    type string;
}
```

When used in the CLI it looks like this:

```
io(config)# server ?
<name:server type [smtp|www|imap]>
io(config)# server
```

It is also possible to modify the help texts for built-in types using the /clispec/$MODE/modifications/type-help element. For example:

```
<typehelp type="unsignedShort">integer</typehelp>
```

16.22. User defined wizards

If our configuration contains large structures that are nontrivial to configure in the CLI, we probably wish to add tailor-made wizards to the CLI which aid the user in the process. Typically we want a wizard to interact with the user, prompt the user, read some responses, and depending on the responses, populate some structures with reasonable default values depending on the user's responses.

We can write our wizards either as executables or as callbacks. In this section we show how to add a user to the AAA namespace using a shell script.

The wizard code needs to be able to perform the following tasks:

- It must be able to interact with the user, i.e. it must be able to read and write to the CLI terminal.
- It must be able to read and write to the db session which is used by the CLI. The CLI will start a db session whenever it enters configuration mode, the wizard code must be able to read and write, not to the database, but to the particular db session which is used by the running CLI. This is done in a shell script using the maapi command which will attach to the current CLI session.

The command to invoke the wizard is added by editing the *confd.cli* file, and recompile it. The workings of the *confd.cli* file is fully described in the UNIX man page *clispec(5)*. A new wizard may be added by adding the following in the confd.cli file:

```
.....
<configureMode>
  .....
  <cmd name="wizard">
    <info>Configuration wizards</info>
    <help>Configuration wizards</help>
    <cmd name="adduser">
      <info>Create a Confd user</info>
      <help>
        Create a Confd user and assign him/her to the proper group
        to control what parts of the system he/she has access to.
      </help>
      <callback>
        <exec>
          <osCommand>./adduser.sh</osCommand
          <args> </args>
        </exec>
      </callback>
    </cmd>
  </cmd>
</configureMode>
```

This way, once inside configuration mode, the command "wizard adduser" will be available. The type of the callback is <exec>, indicating that the wizard is implemented as an executable.

For example, adduser.sh:

```
#!/bin/bash

## Ask for user name
while true; do
    echo -n "Enter user name: "
    read user

    if [ ! -n "${user}" ]; then
 echo "You failed to supply a user name."
    elif maapi --exists "/aaa:aaa/authentication/users/user{${user}}"; then
 echo "The user already exists."
    else
 break
    fi
done

## Ask for password
while true; do
    echo -n "Enter password: "
    read -s pass1
    echo

    if [ "${pass1:0:1}" == "$" ]; then
 echo -n "The password must not start with $. Please choose a "
 echo    "different password."
    else
 echo -n "Confirm password: "
 read -s pass2
 echo

 if [ "${pass1}" != "${pass2}" ]; then
     echo "Passwords do not match."
```

```
 else
     break
 fi
    fi
done

groups=`maapi --keys "/aaa:aaa/authentication/groups/group"`
while true; do
    echo "Choose a group for the user."
    echo -n "Available groups are: "
    for i in ${groups}; do echo -n "${i} "; done
    echo
    echo -n "Enter group for user: "
    read group

    if [ ! -n "${group}" ]; then
 echo "You must enter a valid group."
    else
 for i in ${groups}; do
     if [ "${i}" == "${group}" ]; then
  # valid group found
  break 2;
     fi
 done
 echo "You entered an invalid group."
    fi
    echo
done

echo "Creating user"

maapi --create "/aaa:aaa/authentication/users/user{${user}}"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/password" "${pass1}"

echo "Setting home directory to: /var/confd/homes/${user}"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/homedir" \
            "/var/confd/homes/${user}"

echo "Setting ssh key directory to: /var/confd/homes/${user}/ssh_keydir"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/ssh_keydir" \
            "/var/confd/homes/${user}/ssh_keydir"

maapi --set "/aaa:aaa/authentication/users/user{${user}}/uid" "1000"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/gid" "100"

echo "Adding user to the ${group} group."
gusers=`maapi --get "/aaa:aaa/authentication/groups/group{${group}}/users"`

for i in ${gusers}; do
    if [ "${i}" == "${user}" ]; then
 echo "User already in group"
 exit 0
    fi
done

maapi --set "/aaa:aaa/authentication/groups/group{${group}}/users" \
            "${gusers} ${user}"
```

16.23. User defined wizards in C

We can write precisely the same wizard in C as well. This example makes use of the MAAPI interface, described in the UNIX man page confd_lib_maapi(3) as well as in the user guide chapter "MAAPI - Management Agent API". Thus to fully understand this section, the MAAPI documentation must be read.

Similar to the shell script wizard we must modify the clispec file *"confd.cli"* to indicate the name of a program to execute. We have:

```
<configureMode>
  .....
  <cmd name="wizard">
    <info>Configuration wizards</info>
    <help>Configuration wizards</help>

    <cmd name="cadduser">
      <info>Add a user</info>
      <help>Add a user</help>
      <callback>
        <exec>
          <osCommand>/usr/local/bin/maapi_add_user</osCommand>
          <options>
            <uid>user</uid>
          </options>
        </exec>
      </callback>
    </cmd>
    ....
  </cmd>
</configureMode>
```

Using our modified *"confd.cli"*, a command called "wizard cadduser" will be available in the CLI configuration mode. When this CLI command is executed, the ConfD CLI will invoke the external program called *"/usr/local/bin/maapi_add_user"*.

This program will execute under a pseudo terminal (pty) which is connected to the actual CLI terminal. Thus the external programs invoked by the CLI can manipulate the terminal - not just stdin and stdout.

The invoked program will have two environment variables set which can be used in the MAAPI interface to create an attached MAAPI session towards the currently executing transaction in the CLI. Thus the program must read *"CONFD_MAAPI_USID"* and *"CONFD_MAAPI_THANDLE"* from its environment.

Another thing which is different in the C implementation of a CLI wizard is that the C code must be explicitly aware of which namespace it wants to manipulate. In our case, where we wish to add a user, we wish to manipulate the AAA namespace, *"http://tail-f.com/ns/aaa/1.1"*. The data model defining the AAA namespace is included in a ConfD release and when the data model is compiled a .h file is generated which contains the symbols defined in the namespace. This .h file must be included. Thus:

```
.....

#include "confd_lib.h"
#include "confd_maapi.h"
#include "aaa_bridge.h"

int main(int argc, char **argv)
{
    int msock;
    int debuglevel = CONFD_DEBUG;
    struct in_addr in;
```

```
  struct sockaddr_in addr;
  int usid, thandle;
  char user[255], *pwd, home[255], sshdir[255];
  char buf[BUFSIZ];
  inet_aton("127.0.0.1", &amp;in);
  addr.sin_addr.s_addr = in.s_addr;
  addr.sin_family = AF_INET;
  addr.sin_port = htons(4565);

  confd_init("adduser", stderr, debuglevel);
  confd_load_schemas((struct sockaddr*)&amp;addr,
                     sizeof (struct sockaddr_in));
  if ((msock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
      confd_fatal("Failed to open socket\n");

  if (maapi_connect(msock, (struct sockaddr*)&amp;addr,
                    sizeof (struct sockaddr_in)) < 0)
      confd_fatal("Failed to confd_connect() to confd: %s\n",
                  confd_lasterr());
  usid = atoi(getenv("CONFD_MAAPI_USID"));
  thandle = atoi(getenv("CONFD_MAAPI_THANDLE"));

  if ((maapi_attach2(msock, aaa__ns, usid, thandle)) != CONFD_OK)
      confd_fatal("Failed to attach: %s\n", confd_lasterr());

again0:
  printf("Adding a user\n");
  printf("Username: ");
  if ((fgets(user,255, stdin)) == NULL)
      exit(1);
  user[strlen(user)-1] = 0;
  if (maapi_cd(msock,thandle,"/aaa/authentication/users") != CONFD_OK)
      confd_fatal("cannot CD: %s", confd_lasterr());

  /* check if user exists */
  if (maapi_exists(msock, thandle,"user{%s}", user) == 1) {
      printf("user %s already exists: %s\n", user, confd_lasterr());
      goto again0;
  }

again:
  if ((pwd = getpass("Password: ")) == NULL)
      exit(1);
  strcpy(buf, pwd);
  if ((pwd = getpass("Confirm password: ")) == NULL)
      exit(1);
  if (strcmp(pwd, buf) != 0) {
      printf("Password not confirmed\n");
      goto again;
  }

  printf("Home: ");
  if ((fgets(home,255,stdin)) == NULL)
      exit(1);
  home[strlen(home)-1] = 0;


  printf("SSH dir: ");
  if ((fgets(sshdir, 255, stdin)) == NULL)
      exit(1);
```

```
    sshdir[strlen(sshdir)-1] = 0;



    if (maapi_create(msock,thandle,"user{%s}",user) != CONFD_OK)
        confd_fatal("failed to create user %s: %s\n", user, confd_lasterr());

    if (maapi_set_elem2(msock,thandle,sshdir,"user{%s}/ssh_keydir",user) !=
        CONFD_OK)
        confd_fatal("failed to set ssh keydir: %s\n", confd_lasterr());

    if (maapi_set_elem2(msock,thandle,pwd,"user{%s}/password",user) !=
        CONFD_OK)
        confd_fatal("failed to set password: %s\n", confd_lasterr());

    if (maapi_set_elem2(msock,thandle,home,"user{%s}/homedir",user) !=
        CONFD_OK)
        confd_fatal("failed to set home: %s\n", confd_lasterr());

    printf("user %s added successfully \n", user);
    exit(0);
}
```

The code illustrates several points:

- It reads the above mentioned environment variables and calls `maapi_attach2()` to attach to the transaction.
- The code includes the generated .h file from the AAA namespace. Furthermore, since the code is manipulating the AAA namespace.
- It makes use of the libc function `getpass()` which opens "/dev/tty" to read a password without echoing the characters entered by the user.
- It uses the MAAPI interface to read and write data. Once the program returns, the data written by the program is still not committed. Not until the user executes the **commit** command in the CLI will the data be actually written to the database opened by the CLI, whether running or the candidate.

16.24. User defined commands in C using the C-API

A command can be implemented as a C-callback using the same API as used for actions, with some minor modifications of the input parameters.

The arguments to the command are passed as string params where the first param is the full name of the command, and the remaining params are the arguments that were used when invoking the command from the CLI.

Maapi contains five CLI related functions which can be used for reading and writing to the CLI from inside an action invoked from the CLI - maapi_cli_write, maapi_cli_printf, maapi_cli_prompt, maapi_cli_prompt_oneof, and maapi_cli_read_eof.

Similar to commands implemented as executables, there is an option in the confd.cli file for specifying that the command is implemented through a CAPI callback. It may look like this:

```
<configureMode>
  ...
```

```
    <cmd name="ctest">
      <info>C-API command example</info>
      <help>C-API command example</help>
      <callback>
        <capi>
          <cmdpoint>testcommand</cmdpoint>
        </capi>
      </callback>
    </cmd>
  ...
</configureMode>
```

The capi tag tells confd that the command is implemented using CAPI and the cmdpoint is the name of the action callback to invoke when the command is used from the CLI.

16.25. User defined commands as shell scripts

ConfD comes with a small C program called *maapi*. This program can be used inside shell scripts that are defined as CLI commands, as exemplified in the add_user.sh script above. The *maapi* program is thoroughly described in the man page *maapi(1)*.

The program attaches itself to the current transaction using *maapi_attach()* (see confd_lib_maapi(3)) and executes a single change.

16.26. Modifying built-in commands

There are a number of built-in commands which can be modified in a number of ways. There should not be confused with the auto-rendered commands which cannot be modified in the same way. The auto-rendered commands are derived from the data-model at run-time and does not exist when confd is started. The section below only relates to the built-in commands.

16.26.1. Renaming a command

A built-in command often consists of a command name part and a parameter part. For example, the command **config** in C-mode has the name *config* and takes an optional parameter which can be either *terminal* or *exclusive*.

It is possible to rename the command **config** but not the command argument. I.e. the following is possible,

```
<operationalMode>
  <modifications>
    <move src="config" dest="configure"/>
  </modifications>
</operationalMode>
```

The following will not work since the command is *config* not *config terminal*.

```
<operationalMode>
  <modifications>
    <move src="config terminal" dest="config private"/>
  </modifications>
</operationalMode>
```

In the general case it is difficult to know what are commands and what are arguments to the command. The above could have been defined as two commands **config private** and **config exclusive** without affecting the CLI behavior. Using **confd --cli-c-dump** can be used for determining which part is a command name and which parts are command arguments.

16.26.2. Hiding the old, creating new

If you want to change the way a builtin command works, for example the **config** command above. There is a trick that can be used. It consists of renaming the original command and hiding it. Then add your own command and have it invoke the original hidden command.

Suppose we want to change the config command above to take the parameters *private* and *exclusive* instead of *terminal* and *exclusive*. This is the way to do it:

```
<operationalMode>
  <modifications>
    <move src="config" dest="xxconfig"/>
    <hide src="xxconfig"/>
  </modifications>
    <cmd name="config">
      <info>Manipulate software configuration information</info>
      <help>Manipulate software configuration information</help>
      <callback>
        <exec><osCommand>/usr/local/bin/config.sh</osCommand></exec>
      </callback>
      <params>
        <param>
          <type>
            <enumerate>
              <enum>
                <name>private</name>
                <info>non-locked editing of configuration</info>
              </enum>
              <enum>
                <name>exclusive</name>
                <info>locked editing of configuration</info>
              </enum>
            </enumerate>
          </type>
        </param>
      </params>
    </cmd>
</operationalMode>
```

```
#!/bin/bash

case $1 in
private)
        maapi --clicmd --no-hidden 'xxconfig terminal'
        ;;
shared)
        maapi --clicmd --no-hidden 'xxconfig shared'
        ;;
*)
        maapi --clicmd --no-hidden 'xxconfig terminal'
        ;;
esac
```

16.27. Tailoring show commands

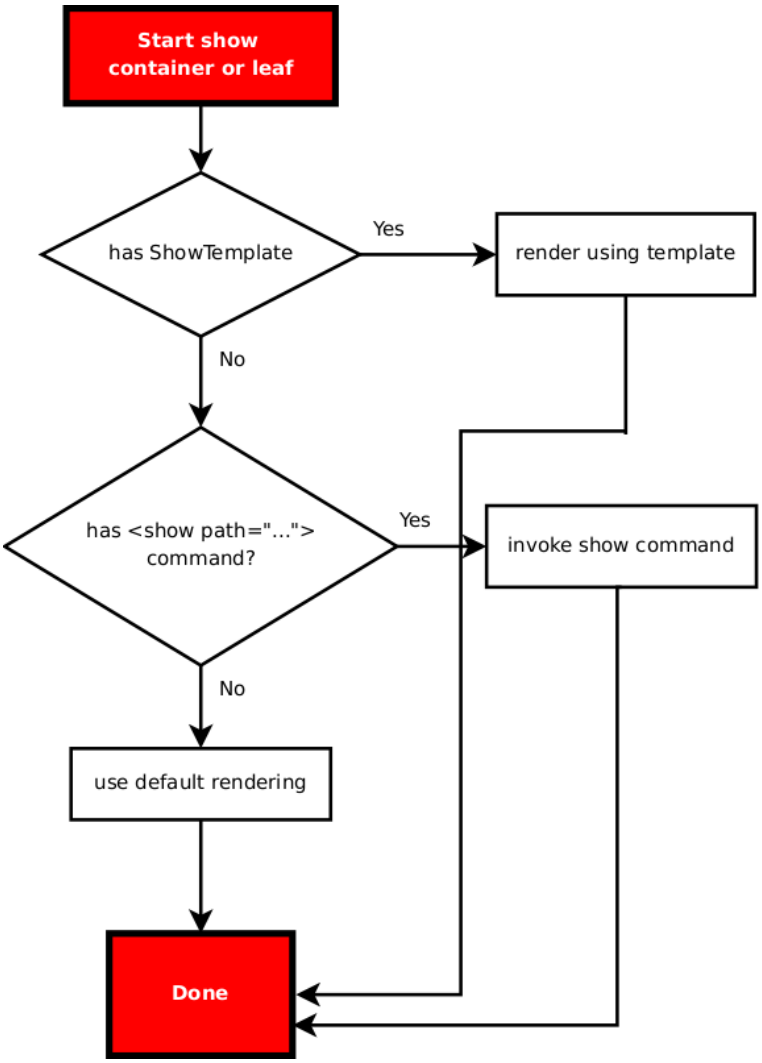
There are several ways show commands can be tailored in the CLI. The data model can be modified, explicit commands can be defined in the clispec files, the auto-rendered commands can be tweaked using

clispec modifications, specific show callbacks can be defined in the clispec files, and show templates can be defined in the clispec files.

16.27.1. How config="false" data is rendered

By default the CLI will create show commands in operational mode for displaying config="false" data. Leaves and containers will be displayed as Name-Value pairs whereas list elements will be displayed as tables, provided that the table will fit on the terminal screen.

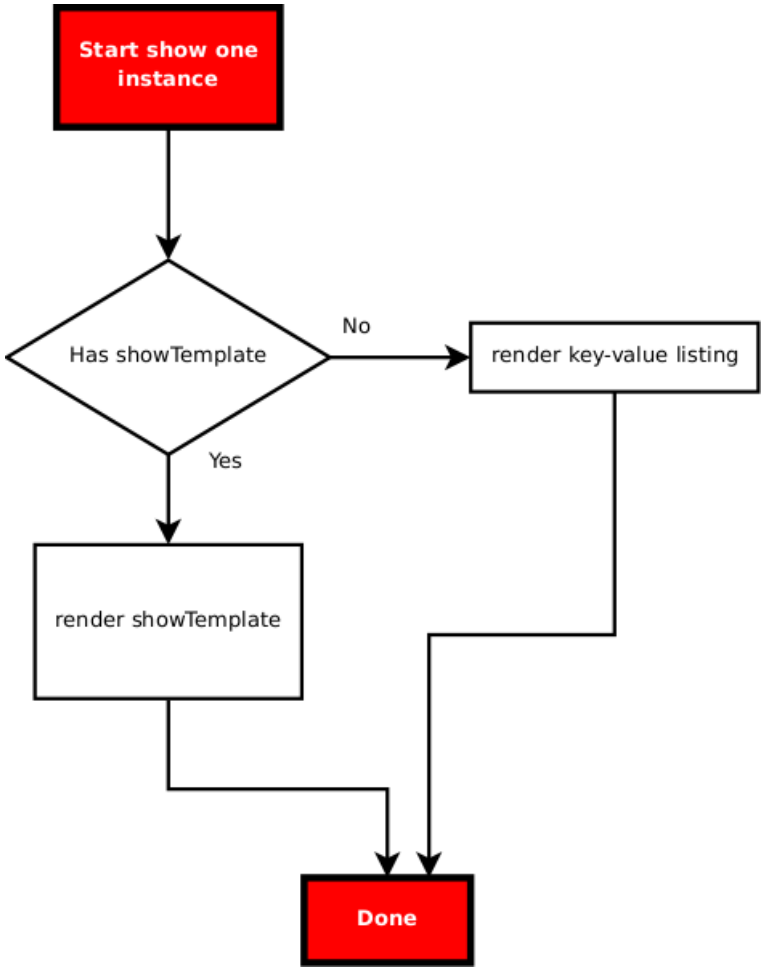
A leaf element is rendered as follows:



Rendering of leaves and containers

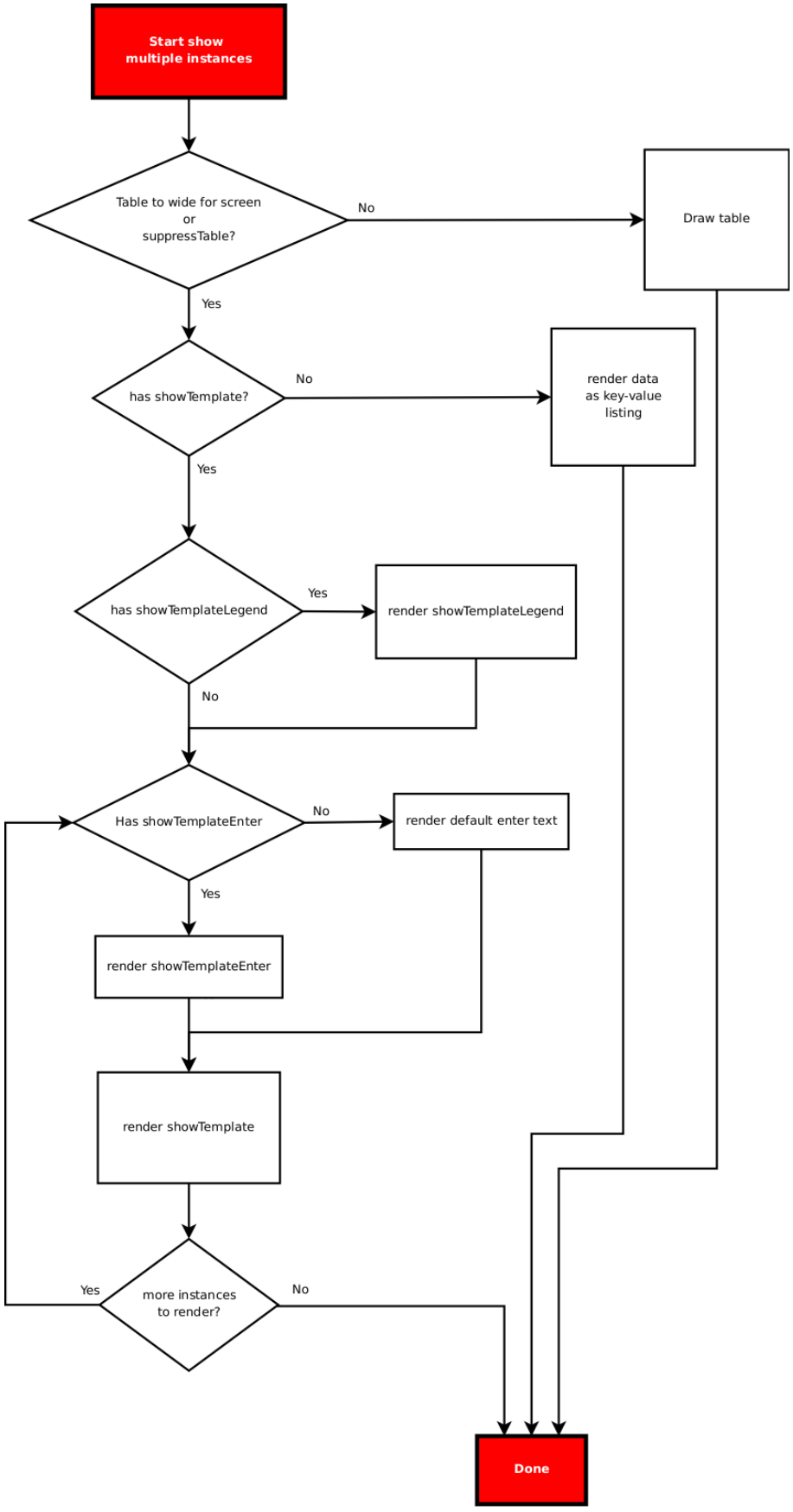
Two different algorithms are used when rendering list elements. One for rendering a single element and one for rendering multiple elements (either the entire table or a subset of the table).

When rendering a single element the following method is used:



Rendering of a single list element

And when rendering a set of list elements the following is used:


Start show multiple instances
Table to wide for screen or suppressTable?
No
Draw table
Yes
has showTemplate?
No
render data as key-value listing
Yes
has showTemplateLegend
Yes
render showTemplateLegend
No
Has showTemplateEnter
No
render default enter text
Yes
render showTemplateEnter
render showTemplate
more instances to render?
Yes
No
Done

Rendering of a set of list elements

16.27.2. Show templates

Show templates can be used in a few different ways. Either as a replacement for the default Name-Value rendering, or for replacing the auto-rendered tables. By default only the Name-Value rendering is replaced when a show template is defined, i.e. the show template will only be invoked for list elements if the table is too wide for the screen (or if table view has been suppressed in some other way).

In order to also replace the default table rendering with the show template the auto-rendered table needs to be suppressed. This is done using the YANG tailf:cli-suppress-table statement in the YANG file.

If a show template is used for rendering a table then the show template needs to render the table header, this is done using a tailf:cli-show-template-legend statement. It also needs to suppress the default *enter* text that is displayed for each instance. This is done by defining an empty tailf:cli-show-template-enter template. Finally, a show template needs to be defined for the list element node that renders each table line. A customized footer can be displayed by using a tailf:cli-show-template-footer statement.

The above will work fine when displaying an entire table. However, when displaying a single instance it might be desirable to either display it as a table line, in which case a table header also needs to be displayed, or in some other way. The problem is that the same show template is used in both situations.

The solution is to add a conditional in the template and display different texts in the two situations. For example (from the `example/cli/show_template/jdemo.yang`):

```
tailf:cli-show-template
       "$(.legend_shown!=true?"
       +"Address         Interface   "
       +"$(.selected~=hwaddr?HW Address         )"
       +"$(.selected~=permanent?Permant  )"
       +"$(.selected~=published?Published)"
       +"$(.selected~=bignum?$(.show_bignum? Bignum ))\n"
       +"============================"
       +"$(.selected~=hwaddr?==================)"
       +"$(.selected~=permanent?=========)"
       +"$(.selected~=published?=========)"
       +"$(.selected~=bignum?$(.show_bignum?========))\n)"
       +"$(ip|ljust:15) $(ifname|ljust:9) - "
       +"$(.selected~=hwaddr?$(hwaddr|ljust:17)  )"
       +"$(.selected~=permanent?$(permanent|ljust:8) )"
       +"$(.selected~=published?$(published|ljust:7)  )"
       +"$(.selected~=bignum?$(.show_bignum?$(.humanreadable? "
```

In the example above the legend is only displayed if it has not already been displayed. This is achieved by inspecting the built-in variable `.legend_shown`. The same behavior can be achieved by using the substatement tailf:cli-auto-legend (see listing below). The example above also tests on the `.selected` variable to determine if each column has been selected to be displayed using the **select** pipe command.

```
tailf:cli-show-template
       "$(ip|ljust:15) $(ifname|ljust:9) - "
       +"$(.selected~=hwaddr?$(hwaddr|ljust:17)  )"
       +"$(.selected~=permanent?$(permanent|ljust:8) )"
       +"$(.selected~=published?$(published|ljust:7)  )"
       +"$(.selected~=bignum?$(.show_bignum?$(.humanreadable? " {
         tailf:cli-auto-legend;
      }
```

It is possible to ask confd to perform a validation of the paths that appear in the templates in the YANG files. This is done by running the command **confd --cli-check-templates** once confd has been started.

16.28. Change password at initial login

To force the user to change the password at initial login, or when the password has expired, a start script can be used.

The start script is specified in the clispec file. Only one start script can be present, if more are present (for example in different clispec files) then only one of them will be executed (unspecified which).

In the clispec file:

```
<start>
  <callback>
   <exec>
   <osCommand>./startup.sh</osCommand>
      <args>$(user)</args>
   </exec>
  </callback>
</start>
```

In startup.sh something along the lines of:

```
#!/bin/bash

user=$1
oldpass=`maapi --get "/aaa:aaa/authentication/users/user{${user}}/password"`

echo "oldpass=${oldpass}"

if [ ${oldpass} == '$1$Dd0v2$Rd899rbrbFTeHuEjAtzvW/' ]; then
     echo "You need to set a new password"

## Ask for password

     while true; do
         echo -n "Enter password: "
         read -s pass1
         echo

         if [ "${pass1:0:1}" == "$" ]; then
            echo -n "The password must not start with $. Please choose a "
            echo    "different password."
         else
            echo -n "Confirm password: "
            read -s pass2
            echo

            if [ "${pass1}" != "${pass2}" ]; then
                echo "Passwords do not match."
            else
                break
            fi
         fi
     done
## create new transaction and write password
```

```
    confd_cmd -r -m -c \
        "mset /aaa:aaa/authentication/users/user{${user}}/password ${pass1}"

fi

echo "Welcome!"
```

Note that the old password needs to be updated to your default startup password, or the test changed into something a bit more sophisticated.

Chapter 17. The SNMP Agent

17.1. Introduction to the ConfD SNMP Agent

The ConfD integrated SNMP agent provides an environment where SNMP and other agents such as NETCONF, Web UI, and CLI, coexist and use the same built-in database (CDB) for configuration storage and the same set of instrumentation functions for controlling managed objects (MOs). Simple bindings from SNMP MIB objects to YANG nodes is all that is needed to open up a configuration database to be accessed by an SNMP manager.

The advantage of having an integrated SNMP agent in ConfD is that configuration data can be accessed directly from the built-in database (CDB) or from user written managed objects without having to do the tedious work of writing a separate set of instrumentation functions just for SNMP. One set of common instrumentation functions is used for serving the NETCONF, CLI, Web UI, and SNMP agents.

confdc --mib2yang is used to make YANG models from MIBs. It also provides the necessary bindings from MIB objects to nodes in the YANG model.

To go the opposite way, from YANG to MIBs, see Section 17.3, “Generating MIBs from YANG”.

The ConfD SNMP agent application provides the following features:

- An extensible SNMP agent that understands SNMPv1, SNMPv2c and SNMPv3.
- A MIB compiler that understands SMIv1 and SMIv2.
- Configuration data is specified in YANG models.
- Common instrumentation functions for controlling managed objects (MO's) via NETCONF, CLI, Web UI, and SNMP agent.
- The SNMP agent uses ConfD AAA datarules to determine access to data, as well as using the standard SNMP view based and user based access control mechanisms.
- The following MIBs are builtin in the ConfD SNMP agent:
  - SNMPv2-MIB RFC 3418
  - SNMP-FRAMEWORK-MIB RFC 3411
  - SNMP-USER-BASED-SM-MIB RFC 3414
  - SNMP-VIEW-BASED-ACM-MIB RFC 3415
  - SNMP-COMMUNITY-MIB RFC 3584
  - SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB RFC 3413
  - SNMP-MPD-MIB RFC 3412
  - TRANSPORT-ADDRESS-MIB RFC 3419
  - SNMP-USM-AES-MIB RFC 3826
  - IPV6-TC RFC 2465
- The following MIBs define the SMI language:

- SNMPv2-SMI RFC 2578
- SNMPv2-TC RFC 2579
- SNMPv2-CONF RFC 2580
- RFC1155-SMI RFC 1155
- RFC-1212 RFC 1212
- RFC-1215 RFC 1215

17.1.1. Implementing MIBs

To set up an SNMP agent to manage objects in the MIB the following information must be provided:

- MIB
- YANG (`.yang` or `.yin`) file.
- Associations between objects in the MIB and nodes in the YANG module.
- Instrumentation functions (not needed for config data if CDB is used)

The MIBs are typically already existing standard or proprietary enterprise MIBs, but they can also be generated from the YANG models.

The MIB compiler needs a mapping between the MIB object to nodes in the YANG module. This is done by adding YANG statements to the data model, that associate YANG nodes with objects in the MIB. The association statements can be written directly in the YANG module file, or in a separate YANG annotation file (see tailf_yang_extensions(5)).

The **confdc** compiler verifies that the types of the SNMP objects and the types in the YANG module are compatible.

There are three main use cases to consider when implementing a MIB with ConfD :

1. The MIB is given, and a YANG module is generated from the MIB.

   The YANG modules and associations are generated with the **confdc --mib2yang** translator program. The generated YANG modules will in this case resemble the structure of the MIBs.

2. The MIB and YANG module are both given (or written manually).

   In this case, MIB associations should be written to bind MIB objects to the nodes in the YANG data model. Statements `tailf:snmp-name`, `tailf:snmp-oid`, etc. are added either directly in the YANG module or in a separate annotation file (see tailf_yang_extensions(5)).

3. The YANG module is given (or written manually), and the MIB is generated from it.

   The MIBs are generated using the **confdc --emit-mib** command.

17.2. Agent Functional Description

The ConfD SNMP agent provides SNMP access to the data available through the ConfD management backplane. The same data can also be accessed via other protocols/applications such as NETCONF, Web UI, and CLI. This data can be stored in CDB, or made available by a data provider.

SNMP has certain features that are not meaningful to model in YANG. There are also some requirements on how the data is to be sorted in tables since SNMP operations require a strict lexicographical order of the elements in a table. Below is a listing of some of the SNMP specific behaviors and what needs to be done:

- The *RowStatus* column in tables are handled by the SNMP agent and must not be part of the YANG model. Rows with a RowStatus column set to 'notReady' or 'notInService' are temporarily stored in the SNMP agent and will not show up in the database. Once activated they will be inserted into the database.

- SNMP requires objects that are stored in tables to be ordered in a strict lexicographical order. If we have a list in a YANG module which is handled by an instrumentation function, the `get_next()` callback function (which must be provided by the user), must return the elements in the same lexicographical order as SNMP expects. If the order of the elements is not correct, then an SNMP manager will not be able to correctly traverse the elements in a table. If the list statement has a `tailf:secondary-index` with the name *snmp*, the agent will iterate over the table using this index. Thus, the instrumentation code can reply with instances in SNMP lexicographical order when the objects are retrieved over SNMP, and a more natural sort order when the objects are retrieved in the CLI and other northbound interfaces.

- Tables with *INDEX* with dynamic length must have a length byte as part of the index. If the table index is specified as *IMPLIED*, then the length byte is excluded from the index. The statement `tailf:sort-order` can be specified in lists and secondary indexes in the YANG module, to control whether index elements should be ordered with the length byte included or not.

- Enumerations must have the same enumerated values in YANG and in the MIB. Note that the symbolic string associated with the enum may be different in the YANG module and MIB.

- SNMP v3 has support for *contexts*. The ConfD SNMP agent uses *""* as the default context where all operations for this context are made towards the *running* database.

- Scalar variables of *TestAndIncr* are automatically handled by the agent.

17.2.1. Operation Overview

The following steps are needed to get a ConfD SNMP agent up and running:

1. Write a MIB module, generate one from a YANG module, or use an existing one.

2. Write a YANG module, generate one from a MIB module, or use an existing one.

3. Write associations that provide the mapping of MIB objects into YANG nodes.

4. Write instrumentation functions for nodes in the YANG module, or store data in CDB.

5. Compile the YANG module into an `.fxs` file.

6. Run the MIB together with the YANG `.fxs` file, and an optional mib annotations file (`.miba`, see mib_annotations(5)), through the MIB compiler to produce a `.bin` file.

7. Configure the agent (`confd.conf` and initial MIB data). Specify which compiled MIBs are to be loaded by the agent (`.bin` files) in `confd.conf` .

8. The produced `.fxs` file as well as the `.fxs` files for the built-in MIBs must be found in the *loadPath* specified in `confd.conf` .

9. Start ConfD .

17.2.2. MIBs and YANG

The basic objects in a MIB are scalar objects and table objects. Each MIB object must be mapped to a node in a YANG module. Scalar MIB objects must be mapped to YANG leafs with matching types, so that the agent can translate the value between the SNMP value and the internal value defined by the YANG type. Tables in the MIB must be mapped to lists in YANG. The mapping between the MIB objects and the YANG nodes is specified in the YANG module (or annotation file for the module) using `tailf:snmp-name` and `tail:snmp-oid` statements. `tailf:snmp-name` specifies the symbolic name of a MIB object, and `tailf:snmp-oid` specifies the corresponding OID.

Let us assume that we have the following MIB named SIMPLE-MIB containing a scalar and table:

```
-- a scalar
numberOfHosts OBJECT-TYPE
    SYNTAX      INTEGER (0..65535)
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "Return the current number of hosts"
    ::= { simpleVariables 1 }

-- a table
hostTable OBJECT-TYPE
    SYNTAX       SEQUENCE OF HostEntry
    MAX-ACCESS   not-accessible
    STATUS       current
    DESCRIPTION
        "The table of hosts."
    ::= { simpleTables 1 }

hostEntry OBJECT-TYPE
    SYNTAX       HostEntry
    MAX-ACCESS   not-accessible
    STATUS       current
    DESCRIPTION
        "Information about a host."
    INDEX       { hostName }
    ::= { hostTable 1 }

HostEntry ::= SEQUENCE {
    hostName              DisplayString,
    hostEnabled           TruthValue,
    hostNumberOfServers   Integer32,
    hostRowStatus         RowStatus
}

hostName OBJECT-TYPE
    SYNTAX      DisplayString
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "The unique index value of a row in this table."
    ::= { hostEntry 1 }

hostEnabled OBJECT-TYPE
    SYNTAX       TruthValue
    MAX-ACCESS   read-create
```

```
    STATUS       current
    DESCRIPTION
        "A bool value saying if host is enabled or not."
    ::= { hostEntry 2 }

hostNumberOfServers OBJECT-TYPE
    SYNTAX       Integer32
    MAX-ACCESS   read-only
    STATUS       current
    DESCRIPTION
        "A read-only integer saying how many servers there currently are."
    ::= { hostEntry 3 }

hostRowStatus OBJECT-TYPE
    SYNTAX       RowStatus
    MAX-ACCESS   read-create
    STATUS       current
    DESCRIPTION
        "The status of this conceptual row in the hostTable."
    ::= { hostEntry 4 }
```

An association must be written to bind the two SNMP objects (the scalar and the table) into YANG. Below is an example of a simple YANG module with SNMP statements that maps to SNMP objects in the MIB.

Example 17.1. Simple YANG module

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

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

  typedef nameType {
    type string {
      length "min .. 255";
    }
  }

  tailf:snmp-mib-module-name TAIL-F-TEST-MIB;

  container simpleObjects {

    leaf numberOfHosts {
      type uint16;
      mandatory true;
      tailf:snmp-name numberOfHosts;
    }

    container hosts {
      list host {
        key name;
        max-elements 64;
        tailf:sort-order snmp;
        tailf:snmp-name hostTable;
```

```
        tailf:snmp-row-status-column 4;

        leaf name {
          type nameType;
        }

        leaf enabled {
          type boolean;
          mandatory true;
          tailf:snmp-name hostEnabled;
        }

        leaf numberOfServers {
          type uint16;
          mandatory true;
          tailf:snmp-oid .3;
        }
      }
    }
  }
}
```

In the code listing above there is one variable `numberOfHosts` mapped to the SNMP scalar variable `numberOfHosts` using the `tailf:snmp-name` statement. The `numberOfServers` object uses the alternative notation with a `tailf:snmp-oid` statement. Which one to use is a matter of taste.

The list *host* is mapped to the SNMP table *hostTable* using `tailf:snmp-name hostTable`. It would also be possible to specify the `tailf:snmp-oid` if preferred. Notice also that for tables which support creation and deletion of rows through a RowStatus column, the statement `tailf:snmp-row-status-column` can be given. (This is not necessary if the model will be used with an existing MIB, but it is necessary for **confdc --emit-mib** to generate a writable table if the model is used for generating a new MIB.) See Section 17.2.9, “The RowStatus column” for more details.

It is possible to map one YANG node to multiple SNMP objects. For example, if an SNMP table augments another table, both these tables can be implemented in a single YANG list, where some leafs are mapped to the base table, and some are mapped to the augmented table.

The following example illustrates the idea. The single YANG list 'interface' is mapped to the MIB tables ifTable, ifXTable, and ipv4InterfaceTable:

```
list interface {
  key index;
  tailf:snmp-name 'ifTable'; // primary table
  tailf:snmp-name 'ifXTable';
  tailf:snmp-name 'IP-MIB:ipv4InterfaceTable';

  leaf index {
    type int32;
  }
  leaf description {
    type string;
    tailf:snmp-name 'ifDescr';  // mapped to primary table
  }
  leaf name {
    type string;
    tailf:snmp-name 'ifXTable:ifName';
  }
```

```
  leaf ipv4-enable {
    type boolean;
    tailf:snmp-name
      'IP-MIB:ipv4InterfaceTable:ipv4InterfaceEnableStatus';
  }
  ...
}
```

17.2.3. Types

When the SNMP agent receives a request to GET an object, it will lookup the object in the compiled MIB, and through the association information find the corresponding YANG node. Next, it will retrieve the correct instances value from a data provider. This value will be encoded according to the type in the YANG module. The SNMP agent translates this value to the corresponding SNMP value, and sends the reply to the manager. For this translation to work, the types in the YANG module and MIB must match.

The following table shows how SMI data types are mapped to YANG datatypes. This mapping is used internally in the agent in runtime, and also by the **confdc --mib2yang** program when a YANG module is generated from a MIB. See the confd_types(3) man page for details about the XSD and confd types.

Table 17.1. SMI mapping to YANG types

SMI	YANG	C value type	XML example
OBJECT IDENTIFIER	yang:object-identifier	C_OID	1.3.6.1.2.1
IpAddress	inet:ipv4-address	C_IPV4	192.168.2.3
Unsigned32	uint32	C_UINT32	
Gauge32	yang:gauge32	C_UINT32	
Counter32	yang:counter32	C_UINT32	
TimeTicks	yang:time-ticks	C_UINT32	
Integer32	int32	C_INT32	
Counter64	yang:counter64	C_UINT64	
INTEGER { enums... }	enumeration	C_ENUMHASH	udp
INTEGER	int32	C_INT32	42
DisplayString	string	C_BUF/C_STR	Hello world!
OCTET STRING (with DISPLAY-HINT on the form "Na" or "Nt")	string	C_BUF/C_STR	Hello world!
OCTET STRING (binary), Opaque	tailf:hex-list	C_BINARY	4f:12:ff
IPV6-TC::Ipv6Address	inet:ipv6-address	C_IPV6	::213.180.94.158
SNMPv2-TC::DateAndTime	yang:date-and-time	C_DATETIME	2006-08-17T16:30:53+02:00
SNMPv2-TC::TruthValue	boolean, enumeration (1), empty	C_BOOL, C_ENUMHASH	true
SNMPv2-TC::PhysAddress	yang:phys-address	C_BINARY	
SNMPv2-TC::MacAddress	yang:mac-address	C_BINARY	

SMI	YANG	C value type	XML example
SNMPv2-TC::TimeStamp	yang:timestamp	C_UINT32	
SNMPv2-TC::TimeInterval	int32	C_INT32	
SNMPv2-TC::TAddress	tailf:octet-list	C_BINARY	

(1) SNMPv2-TC::TruthValue is a bit special. At runtime, the agent can map it either to a normal enumeration (which is how it is defined in SNMPv2-TC), to a boolean, to an empty leaf, or to a presence-container. When **confd --mib2yang** is used to create the YANG module from a MIB, it uses the enumeration mapping. This is also the recommended mapping. If a boolean is used, it cannot be part of the INDEX in a table.

The following table shows how YANG data types are mapped to SMI datatypes. This mapping is used internally in the agent in runtime, and also by the **confdc --emit-mib** program when a MIB is generated from a YANG module.

If the association between the MIB and YANG module is written manually, the type mappings in this table must be used.

Some of the more complex YANG types that cannot be easily translated to native SMI types are translated into strings of the type "ConfdString" In this case, the human-readable string value is passed over SNMP. These types cannot be used as INDEX in SNMP tables.

See the confd_types(3) man page for details about the XSD and confd types.

Table 17.2. YANG mapping to SMI types

YANG	SMI	C value type	Use as INDEX
int32	Integer32	C_INT32	yes
int16	Integer32 (-32768..32767)	C_INT16	yes
int8	Integer32 (-128..127)	C_INT8	yes
uint64	ConfdString	C_UINT64	yes
uint32	Unsigned32	C_UINT32	yes
uint16	Unsigned32 (0..65535)	C_UINT16	yes
uint8	Unsigned32 (0..255)	C_UINT8	yes
boolean	SNMPv2-TC::TruthValue	C_BOOL	no
enumeration	INTEGER { enums... }	C_ENUMHASH	yes
string	OCTET STRING	C_BUF / C_STR	yes
decimal64	ConfdString	C_DECIMAL64	no
int64	ConfdString	C_INT64	no
union	ConfdString	depending on type	no
binary	OCTET STRING	C_BINARY	no
empty	SNMPV2-TC::TruthValue	C_BOOL	no
identity	not supported	n/a	n/a

YANG	SMI	C value type	Use as INDEX
yang:date-and-time	SNM-Pv2-TC::DateAndTime	C_DATETIME	yes
yang:counter32	Counter32	C_UINT32	yes
yang:counter64	Counter64	C_UINT64	yes
yang:gauge32	Gauge32	C_UINT32	yes
yang:object-identifier	OBJECT IDENTIFIER	C_OID	yes
xs:float, xs:double, xs:decimal	ConfdString	C_DOUBLE	no
inet:ipv4-address	IpAddress	C_IPV4	yes
inet:ipv6-address	IPV6-TC::Ipv6Address	C_IPV6	yes
inet:ip-address	OCTET STRING (SIZE (4|16))	C_IPV4 | C_IPV6	yes
inet:host	ConfdString	C_BUF / C_STR	no
inet:domain-name	ConfdString	C_BUF / C_STR	no
inet:port-number	Unsigned32 (0..65535)	C_UINT16	yes
inet:ipv4-prefix	OCTET STRING (SIZE (5))	C_IPV4PREFIX	yes
inet:ipv6-prefix	OCTET STRING (SIZE (17))	C_IPV6PREFIX	yes
inet:ip-prefix	OCTET STRING (SIZE (5|17))	C_IPV4PREFIX | C_IPV6PREFIX	yes
tailf:size	OCTET STRING	C_BUF / C_STR	no
tailf:octet-list, tailf:hex-list	OCTET STRING	C_BINARY	yes
xs:date	ConfdString	C_DATE	no
xs:time	ConfdString	C_TIME	no
xs:duration	ConfdString	C_DURATION	no
xs:hexBinary	OCTET STRING	C_BINARY	no
xs:QName	not supported	n/a	n/a

A YANG presence container and a leaf of type empty can be mapped to a SMI scalar or columnar object of type SNMPv2-TC::TruthValue. If the empty leaf or presence container exists, the SMI object is 'true', and if it does not exist, but its parent exists, it has the value 'false'. Setting the SMI object to 'true' creates the leaf or presence container, and setting it to 'false' deletes it.

17.2.4. Generating the YANG module

The **confdc --mib2yang** is used to generate a YANG (`.yang`) files from MIBs. The element structure in the resulting YANG module will resemble the structure of the objects in the MIB.

If the MIB IMPORTs other MIBs, these MIBs must be available (as .mib files) to the compiler when a YANG module is generated. By default, all MIBs in the current directory and all builtin MIBs (see Section 17.1, “Introduction to the ConfD SNMP Agent”) are available. Since the compiler uses the tool **smidump** to perform the conversion to YANG, the environment variable `SMIPATH` can be set to a colon-separated list of directories to search for MIB files.

Example 17.2. Generating and compiling YANG from MIB

```
$ confdc --mib2yang -o SIMPLE-MIB.yang SIMPLE-MIB.mib
$ confdc -c -o SIMPLE-MIB.fxs SIMPLE-MIB.yang
```

Below is the generated YANG module. The structure of the YANG module resembles the structure of the SIMPLE-MIB it was generated from.

Example 17.3. The YANG file generated by confdc --mib2yang

```
module SIMPLE-MIB {
  namespace "http://tail-f.com/ns/mibs/SIMPLE-MIB/200702080000Z";
  prefix SIMPLE-MIB;
  tailf:id "";
  tailf:snmp-mib-module-name SIMPLE-MIB;

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

  import SNMPv2-TC {
    prefix SNMPv2-TC;
  }

  revision 2007-02-08 {
    description "";
  }
  container SIMPLE-MIB {
    container variables {
      tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.1;
      leaf numberOfHosts {
        type int32;
        tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.1.1;
      }
    }
    container hostTable {
      list hostEntry {
        key hostName;
        tailf:sort-order snmp;
        tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.2.1;
        leaf hostName {
          type hostNameType;
          tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.2.1.1.1;
        }
        leaf hostEnabled {
          type SNMPv2-TC:TruthValue;
          tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.2.1.1.2;
        }
```

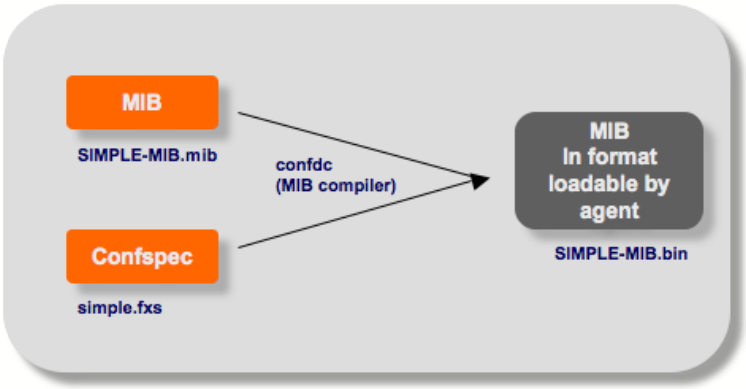
```
        leaf hostNumberOfServers {
          type int32;
          tailf:snmp-oid 1.3.6.1.4.1.24961.3.1.2.1.1.3;
        }
      }
    }
  }
  typedef hostNameType {
    type string {
      length "min .. 64";
    }
  }
}
```

17.2.5. Compiling the YANG modules

Compile the YANG modules representing MIBs the same way that any other YANG module is compiled, using **confdc** .

Note that all YANG modules representing the builtin MIBs are available in `$CONFD_DIR/src/confd/snmp/yang` directory. The parameter **--yangpath** can be given to the compiler to search this directory.

17.2.6. Compiling the MIBs



MIB compilation

The **confdc** compiler is used for compiling the MIB into a binary format that can be loaded by the ConfD SNMP agent. The MIB is compiled with the YANG .fxs file with associations that map the YANG nodes into objects in the MIB. The resulting file is a binary (.bin) file that is loaded into the ConfD SNMP agent.

```
$ confdc -c SIMPLE-MIB.mib simple.fxs
```

If the MIB IMPORTs other MIBs, these MIBs must be available (as compiled .bin files) to the compiler. By default, all compiled MIBs in the current directory and all builtin MIBs are available. Use the parameters *--include-dir* or *--include-file* to specify where the compiler can find the compiled MIBs.

Note that not every object in the MIB must have a mapping to a node in the YANG module. By using a separate MIB annotation file, ConfD can be instructed how these missing objects should be treated by the SNMP agent. The agent can treat the objects either as not implemented, or as implemented but non-existent.

The format of an annotation line is *object-name annotation*, where *object-name* is the name of an object type (column or scalar), and *annotation* has the form `behavior=noSuchObject`, `behavior=noSuchInstance`, `max_access=not_accessible`, `max_access=read_only`.

If a line is blank or starts with a # character, it is ignored. An object name may occur on several lines.

Example:

```
$ cat HOST-RESOURCES-MIB.miba
# tell the agent to not implement these objects
hrPartitionID           behavior=noSuchInstance
hrSWInstalledDate       behavior=noSuchObject
$ confdc -c HOST_RESOURCES-MIB.mib \
         --mib-annotation HOST_RESOURCES.miba host-resources.fxs
```

17.2.7. Loading MIBs

The ConfD SNMP agent have the following built-in MIBs:

- SNMPv2-MIB, a mandatory MIB for an agent. This MIB is always loaded at start-up.
- SNMP-MPD-MIB, a mandatory MIB for an agent. This MIB is always loaded at start-up if the agent is configured for SNMPv3.
- SNMP-FRAMEWORK-MIB, a mandatory MIB for an agent. This MIB is always loaded at start-up if the agent is configured for SNMPv3.
- SNMP-COMMUNITY-MIB, handles SNMP v1 and v2c communities.
- SNMP-VIEW-BASED-ACM-MIB, handles the view based access control.
- SNMP-USER-BASED-SM-MIB, handles the user based access control.
- SNMP-TARGET-MIB, to be able to configure targets for SNMP traps.
- SNMP-NOTIFICATION-MIB, defines SNMP traps.
- IPV6-TC, defines some IPv6 specific TEXTUAL-CONVENTIONs.
- TRANSPORT-ADDRESS-MIB, defines some OBJECT-IDENTITYs for transport protocols. This MIB module cannot be loaded as a built-in module in the agent. If some other MIB IMPORTs this MIB, then it needs to be compiled and loaded as other non-built-in MIBs.
- SNMP-USM-AES-MIB, defines an OBJECT-IDENTITY for the AES privacy protocol. This MIB module must not be loaded into the agent.

These MIBs of course must have their corresponding YANG .fxs files loaded in order for the SNMP agent to work (see the next section). The MIBs themselves are not required to be loaded. The user decides which MIBs should be loaded, and there may be reasons to not provide SNMP access to certain MIBs.

The MIBs SNMPv2-MIB, SNMP-MPD-MIB, and SNMP-FRAMEWORK-MIB, are always loaded into the ConfD SNMP agent at start-up. These MIBs are required for an SNMP agent.

The other built-in MIBs are not loaded per default, which means that they cannot be accessed/configured via SNMP but of course via for example CDB init files (see Section 5.8, “Loading initial data into CDB”) NETCONF, or even CLI directly. If the intention is to have these MIBs loaded, they must be listed in `confd.conf` without any explicit paths to where they are stored as shown in the example below.

Other MIBs (that are not built-in) are loaded by specifying their absolute or relative paths, or alternatively the MIBs can be loaded from ConfDs loadPath. We recommend that these MIBs are loaded from the load path. This is how other data model files (`.fxs` etc) are handled.

The main reason for this recommendation is how MIBs can be dynamically reloaded. MIBs that are explicitly listed are reloaded by giving the command **confd --reload**. If any MIB cannot be loaded for whatever reason, ConfD halts. MIBs in the load path are reloaded using the data model upgrade functions, see Chapter 13, *In-service Data Model Upgrade*.

Example 17.4. Specifying built-in MIBs to be loaded into the agent

```
<snmpAgent>
  <enabled>true</enabled>
  <mibs>
    <!-- Load built-in MIBS -->
    <file>SNMP-COMMUNITY-MIB.bin</file>
    <file>SNMP-VIEW-BASED-ACM-MIB.bin</file>
    <file>SNMP-USER-BASED-SM-MIB.bin</file>
    <file>SNMP-TARGET-MIB.bin</file>
    <file>SNMP-NOTIFICATION-MIB.bin</file>

    <!-- Load specific user MIBs -->
    <file>/etc/confd/mibs/SIMPLE-MIB.bin</file>

    <!-- Load all MIBs present in the loadPath -->
    <fromLoadPath>true</fromLoadPath>
  </mibs>
</snmpAgent>
```

17.2.8. Loading YANG modules for built-in MIBs

The SNMP agent has a few built-in MIBs that store its configuration data in CDB. The following `.fxs` files defines namespaces for the built-in MIBs and must be loaded at start-up if the SNMP agent is enabled:

- `SNMPv2-MIB.fxs`, `SNMPv2-SMI.fxs`, `SNMPv2-TC.fxs` - contains base SNMPv2 types
- `SNMP-FRAMEWORK-MIB.fxs`
- `SNMP-MPD-MIB.fxs`
- `SNMP-COMMUNITY-MIB.fxs`
- `SNMP-VIEW-BASED-ACM-MIB.fxs`
- `SNMP-USER-BASED-SM-MIB.fxs`
- `SNMP-TARGET-MIB.fxs`
- `SNMP-NOTIFICATION-MIB.fxs`

Preferably a `loadPath` in the `confd.conf` file can be set to the directory where these files are installed. If ConfD fails to load these files it will terminate with a fatal error.

The built-in `.fxs` files are delivered as pre-built, but the YANG source code is provided as well. Tampering with these files is not advised and may result in a serious internal error. However we may wish

to recompile these YANG modules using the **confdc** compiler flag `--export`, to not expose the built-in MIB data to other ConfD internal protocols/applications such as NETCONF, CLI, and Web UI. The YANG source code is provided for this reason. Also it is possible to provide external callpoints for the built-in MIB data to store the data in an external database instead of CDB.

17.2.9. The RowStatus column

The rowstatus column for tables is always handled by the SNMP agent and should not be modeled in the YANG modules. The `tailf:snmp-row-status-column` statement can be left out and the row status column will be looked up by the compiler.

The *RowStatus* column in an SNMP table is used for reading the status of a conceptual row in a table and for creating and deleting new conceptual rows in the table. The following values are always defined for the row status:

- *active* (1) - indicates that the conceptual row is available.
- *notInService* (2) - indicates that the conceptual row is unavailable. This is a temporary state where the row is stored in the SNMP agent and not in the database. A row with a RowStatus of 'notInService' can be made 'active' which means that the row will be inserted into the database.
- *notReady* (3) - indicates that the conceptual row is missing information. This is a temporary state where the row is stored in the SNMP agent and not in the database. A RowStatus of 'notReady' is returned to indicate that the row is missing a value for one or more mandatory column(s). When the row have all the mandatory values set, a RowStatus of 'notInService' will be returned instead of 'notReady'.
- *createAndGo* (4) - set by manager to create new row instance.
- *createAndWait* (5) - set by manager to create new row instance but not make it available. A RowStatus of 'notInService' or 'notReady' is returned depending on if all values for the mandatory columns are set or not.
- *destroy* (6) - set by manager to delete all instances in the conceptual row.

The *createAndWait* creates a new instance of a conceptual row in a temporary state where the row will have a RowStatus set to 'notReady' or 'notInService' depending on if all the mandatory columns are set for the column or not. It can be made 'active' and will then be inserted into the database. Note that there are no guarantees that the row will exist more than 5 minutes (by default) in the temporary storage in the SNMP Agent. The temporary cache used by the SNMP agent for this storage is volatile. The temporary storage time can be configured in by setting *temporaryStorageTime* in `confd.conf` .

To delete a conceptual row the *destroy* value is used.

17.2.10. TestAndIncr

When a YANG module is generated from a MIB, and the MIB contains any scalar object of type *TestAndIncr*, these objects are not translated into the YANG module, since they don't make sense outside SNMP. Then, when the MIB is compiled, the compiler generates code so ConfD automatically handles these objects. No coding is required.

17.2.11. MIB access and YANG config

Objects in MIBs can be read-only or writable. In YANG, nodes are marked as representing configuration or non-configuration data.

Read-only MIB objects

If a MIB object is read-only, it can be mapped to a configuration or non-configuration YANG node.

When a YANG module is generated from a MIB, all read-only MIB objects are translated into non-configuration YANG nodes.

Writable MIB objects

If a MIB object is writable, it is usually mapped to a configuration YANG node. However, in some cases, the MIB object doesn't really represent configuration, but is rather writable operational data. An example could be a scalar variable `rebootRouter`. When written to, the router will reboot. In order to support this, non-configuration YANG nodes can be marked with `tailf:writable true`. Writable MIB objects can be mapped into non-configuration YANG nodes that are marked with `tailf:writable true`.

When a YANG module is generated from a MIB, writable MIB objects are translated into configuration YANG nodes, unless the MIB object is marked as representing writable operational data in a MIB annotation file (see mib_annotations(5)).

If the SNMP agent receives a SET PDU with one or more writable operational objects, it will start a read-write transaction towards `CONFD_OPERATIONAL`. In this transaction, the agent will write all variables from the PDU, and then it will commit the transaction. Instrumentation code needs to be written to handle these writes.

See Section 7.8, “Writable operational data” for how these objects are implemented, and `examples.confd/snmpa/9-writable-operational` in the ConfD examples collection for an example of how this can be implemented.

17.2.12. Optional YANG nodes

There is no protocol support in SNMP to delete optional nodes. Conceptual table rows are typically created and deleted by using a `RowStatus` column, but there is no standardized way to delete optional nodes. One technique to handle this is to introduce a special value for the object, so that the object is deleted when it is set to this special value. ConfD supports this technique with the YANG extension `tailf:snmp-delete-value`.

In order to use this technique, an optional leaf in the YANG model is mapped to a scalar or columnar object in the MIB module. The data type definition of the MIB object allows the same values as the corresponding YANG leaf, and in addition it also allows one extra value, not allowed by the YANG leaf. This special value is also defined in the YANG model in the `tailf:snmp-delete-value` statement.

In the following example, we define a MIB object `fooOptionalLeaf`, and corresponding YANG leaf `foo-optional-leaf`.

Example 17.5. SMI definition of an optional object

```
fooOptionalLeaf OBJECT-TYPE
    SYNTAX       Integer32 (0..255)
    MAX-ACCESS   read-create
    STATUS       current
    DESCRIPTION
        "The special value zero means not used."
    ::= { fooEntry 3 }
```

Example 17.6. YANG definition of an optional leaf

```
leaf foo-optional-leaf {
  type int32 {
    range "1..255";
  }
  tailf:snmp-delete-value 0;
  tailf:snmp-name fooOptionalLeaf;
}
```

When the SNMP agent receives a SET request to set this object to '0', the leaf will be deleted.

If the `tailf:snmp-delete-value` statement has the substatement `tailf:snmp-send-delete-value`, the same special value will be returned on a GET request, instead of the default `noSuchInstance`.

17.2.13. tailf:sort-order on tables

Tables in SNMP are strictly lexicographically ordered. An SNMP table is typically traversed with `GET-NEXT` requests, where given a previous index of a row the next greater index is returned. Since the table is specified in a YANG module and may be stored in an external database or perhaps as a managed object (MO) written in C, it is important that the `get_next()` function returns the elements in correct order. If the *get-next* function doesn't return the elements properly in order, SNMP will not work. If CDB is used to store the data the ordering of the elements will be correct. Note that the `tailf:sort-order` statement may have to be specified for indexes with dynamic length (see Section 17.2.8, “Loading YANG modules for built-in MIBs”).

Types with dynamic length in SNMP like OCTET STRING will have a length indicator when they are part of the *INDEX*, so the ordering for strings stored in such a table will be on length first, unless they are declared as IMPLIED as in IMPLIED OCTET STRING. In this case the index will not have any length indicator included, and the table should be sorted as normal.

The following values can be given to the `tailf:sort-order` statement:

`normal`	This is the default and means that the table is sorted using the key values. This value should be used when the corresponding SNMP table does not have any INDEX object with dynamic length.
`snmp`	This value means that when sorting, any key element of dynamic length will have the length prepended to the value before sorting. It should be used if the corresponding SNMP table has any INDEX object with dynamic length, and no IMPLIED object.
`snmp-implied`	This value is the same as *`snmp`*, except that the last key element will not prepend the length indicator to the key value. It should be used if the corresponding SNMP table has any IMPLIED INDEX.

Here's an example of a MIB table with a DisplayString with dynamic length as index.

Example 17.7. simple.mib

```
hostTable OBJECT-TYPE
    SYNTAX       SEQUENCE OF HostEntry
    MAX-ACCESS   not-accessible
    STATUS       current
```

```
    DESCRIPTION
        "The table of hosts."
    ::= { simpleTables 1 }

hostEntry OBJECT-TYPE
    SYNTAX       HostEntry
    MAX-ACCESS   not-accessible
    STATUS       current
    DESCRIPTION
        "Information about a host."
    INDEX       { hostName }
    ::= { hostTable 1 }

HostEntry ::= SEQUENCE {
    hostName             DisplayString,
    hostEnabled          TruthValue,
    hostNumberOfServers  INTEGER,
    hostRowStatus        RowStatus
}
```

If the list corresponding to this SNMP table is stored in CDB, the definition in the YANG module must specify `tailf:sort-order snmp` so that the table is sorted correctly (with length indicator included).

Example 17.8. simple.yang

```
list host {
  key name;

  tailf:sort-order snmp;

  leaf name {
    type nameType;
  }

  leaf enabled {
    type boolean;
    mandatory true;
  }

  leaf numberOfServers {
    type uint16;
    mandatory true;
  }
}
```

When the sort order is set to "snmp" or "snmp-implied" on a list, it affects the displayed sort order in all northbound interfaces. Thus the list of hosts above will be sorted according to SNMP lexicographical ordering, even in e.g. the CLI. Sometimes this may be confusing to users. This problem can be solved by adding a `tailf:secondary-index` to the list:

Example 17.9. simple.yang with secondary index

```
list host {
  key name;

  tailf:secondary-index snmp {
```

```
    tailf:index-leafs "name";
    tailf:sort-order snmp;
  }

  leaf name {
    type nameType;
  }

  leaf enabled {
    type boolean;
    mandatory true;
  }

  leaf numberOfServers {
    type uint16;
    mandatory true;
  }
}
```

When the SNMP agent traverses a table, it checks if there is a secondary-index with the reserved name "snmp" defined for the table. If there is such an index, the agent traverses the table using this index.

In the example above, the `host` table is sorted in normal order, which means that *"arthur"* appears before *"ford"*. But since there is a secondary-index called *snmp*, the SNMP agent will use this index when traversing the table, so that *4."ford"* appears before *5."arthur"* over SNMP.

Note that the presence of a *secondary-index* in the YANG module is not enough; the instrumentation code must be prepared to perform the actual sorting. See confd_lib_dp(3) for details.

17.2.14. Enumerations

Enumerations in SNMP have textual representation mapped to an integer. A simple example would be the definition for *TruthValue*:

Example 17.10. TruthValue from the SNMPv2-TC

```
TruthValue ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "Represents a boolean value."
    SYNTAX       INTEGER { true(1), false(2) }
```

The **confdc --mib2yang** tool produces the following type definition for *TruthValue*:

Example 17.11. A typedef for TruthValue

```
typedef TruthValue {
  type enumeration {
    enum true {
      value 1;
    }
    enum false {
      value 2;
    }
  }
```

```
}
```

17.2.15. Notifications

Notifications are defined by the NOTIFICATION-TYPE macro in SMIv2. There are two types of notifications in SNMP: *trap* and *inform*. When the managed object needs to send *trap* notifications the following functions should be called (from MO's written in C).

Example 17.12. Functions for sending notification from C

```
int confd_register_snmp_notification(
    struct confd_daemon_ctx *dx, int fd,
    const char *notify_name, const char *ctx_name,
    struct confd_notification_ctx **nctx);

int confd_notification_send_snmp(
    struct confd_notification_ctx *nctx, const char *notification,
    struct confd_snmp_varbind *varbinds, int num_vars);
```

The `confd_register_snmp_notification()` function is typically called once to register the parameters common to a set of notifications, and then the individual notifications are sent by calling `confd_notification_send_snmp()`. The daemon context pointer `dx` is obtained by calling `confd_init_daemon()`, and the socket file descriptor `fd` is a worker socket connected to the ConfD daemon via a call to `confd_connect()`. See confd_lib_dp(3) man page for details about these functions. Note also that a control socket must be connected to the ConfD daemon before calling `confd_register_snmp_notification()`.

The `notify_name` parameter matches the NotifyName in the snmpNotifyTable in the SNMP-NOTIFICATION-MIB. Trap will be sent to targets pointed out by NotifyName. If NotifyName is *""*; the normal procedure defined in SNMP-NOTIFICATION-MIB is used, i.e. the trap is sent to all managers. Otherwise, the NotifyName is used to find an entry in the SnmpNotifyTable which defines how to send the notification (as an Inform or a Trap), and to select targets from SnmpTargetAddrTable (using the Tag).

The `ctx_name` is the name of the context. The default context is *""*.

The `notification` string is the notification name. For example `"coldStart"` or `"warmStart"`. This symbolic name of a notification must be defined in a MIB that is loaded into the agent.

If the empty string is used as notification name, the notification to send is constructed from the `varbinds` array alone, which must then contain a value for the *snmpTrapOID* variable.

The `varbinds` array contains variable bindings for parameters that should be sent along in the notification. The include file `confd_lib.h` defines data structures for these:

Example 17.13. SNMP varbind structures from `confd_maapi.h`

```
enum confd_snmp_var_type {
    CONFD_SNMP_VARIABLE = 1,
    CONFD_SNMP_OID      = 2,
    CONFD_SNMP_COL_ROW  = 3
};

struct confd_snmp_oid {
    int oid[128];
```

```
    int len;
};

struct confd_snmp_col_row {
    char column[256];
    struct confd_snmp_oid rowindex;
};

struct confd_snmp_varbind {
    enum confd_snmp_var_type type;
    union {
        char name[256];
        struct confd_snmp_oid oid;
        struct confd_snmp_col_row cr;
    } var;
    confd_value_t val;
};
```

Each *varbind* is either:

a variable	a symbolic name of a scalar variable referred to in the notification specification.
a column	a symbolic name of a column variable. Rowindex is the index of the specified column.
an OBJECT IDENTIFIER	for the instance of an object, scalar variable or column variable.

If a value is given it will be used. If it is not given (i.e. set to `C_NOEXISTS`) then the agent will look up the value with a `GET` operation.

Example 17.14. Notification registration

```
int setup(struct confd_daemon_ctx *dctx, int workersock,
          struct confd_notification_ctx **nctx)
{
    if (confd_register_snmp_notification(dctx, workersock,
                                         "std_v1_trap", "", nctx)) != CONFD_OK)
        return CONFD_ERR;
    return confd_register_done(dctx);
}
```

Example 17.15. Sending a coldStart notification

```
int test1(struct confd_notification_ctx *nctx)
{
    return confd_notification_send_snmp(nctx, "coldStart", NULL, 0);
}
```

Example 17.16. Sending a notification with a varbind

```
int test2(struct confd_notification_ctx *nctx)
{
```

```
    struct confd_snmp_varbind vb;
    vb.type = CONFD_SNMP_VARIABLE;
    strcpy(vb.var.name, "myVariable");
    CONFD_SET_INT32(&vb.val, 32);
    return confd_notification_send_snmp(nctx, "notif1", &vb, 1);
}
```

The *inform* type notifications are similar to the *trap* type except there is an acknowledgment sent back from the manager that received the inform. Two callbacks needs to be registered to receive the result of the inform, and there's a separate function for sending an inform.

```
int confd_register_notification_snmp_inform_cb(
    struct confd_daemon_ctx *dx,
    const struct confd_notification_snmp_inform_cbs *cb);
```

```
int confd_notification_send_snmp_inform(
    struct confd_notification_ctx *nctx, const char *notification,
    struct confd_snmp_varbind *varbinds, int num_vars,
    const char *cb_id, int ref);
```

The struct confd_notification_snmp_inform_cbs is defined as:

```
struct confd_notification_snmp_inform_cbs {
    char cb_id[MAX_CALLPOINT_LEN];
    void (*targets)(struct confd_notification_ctx *nctx,
                    int ref, struct confd_snmp_target *targets,
                    int num_targets);
    void (*result)(struct confd_notification_ctx *nctx,
                   int ref, struct confd_snmp_target *target,
                   int got_response);
    void *cb_opaque;        /* private user data */
};
```

In order to debug the notification sending process in ConfD, the `/confdConfig/logs/developerLogLevel` in confd.conf(5) can be set to "trace".

17.3. Generating MIBs from YANG

The previous sections have discussed the scenario where there is an existing set of MIB files, and then **confdc --mib2yang** is used to convert these to YANG with the associations that the agent needs.

If instead no MIBs exist, but a number of YANG files (complied to `.fxs` files), these can be translated to MIB files (in SMIv2 syntax), using the `--emit-mib` option of **confdc**.

The normal operation of the translator is to select those nodes that have an `tailf:snmp-oid` statement, and ignore the others. If the option `--generate-oids` is given (described later in this section), all elements are selected, unless explicitly marked with `tailf:snmp-exclude-object`.

The value of the `tailf:snmp-oid` statement can be either a one-component suffix, for example ".4", or a full OID, such as "1.3.6.1.4.1.12345" or "private.1.12345". If it's a suffix, a full OID should be specified for some ancestor element, in the YANG module header, or using the `--oid` option.

In order to be selected for translation to the MIB file, an element must also match the module name. The module name can be given as an `tailf:snmp-mib-module-name` statement in the YANG module header, which is then inherited by all nodes, or using the `--module` option. It can also be specified on a node, which then overrides the value from the header or ancestor nodes.

Since tables can not reside inside tables in SMI, lists containing lists are handled by moving the inner lists up to top level.

Nodes inside containers in lists are given OIDs directly below the table entry, and names which are the concatenation of the path down from the table level. The containers should not have an OID.

If a RowStatus column is desired for a table, use the statement `tailf:snmp-row-status-column` on the corresponding list. The statement's value should be the column number (i.e., the last OID component, with no leading period). The object will be called `rowstatus`.

17.3.1. Translating a `.fxs` file to a MIB

The form of the translation command is shown below (using fictitious parameters):

confdc --emit-mib FOO-MIB.mib --oid enterprises.24961 -- foo.fxs

The basename of the output file (here, `FOO-MIB`) by default becomes the name of the module (with all letters made upper case). The option `--module` can be used to specify the module name.

Any other `.fxs` files we depend on have to be given with `-f`, as usual.

confdc --emit-mib FOO-MIB.mib --oid enterprises.24961 -f types.fxs -- foo.fxs

To build the `.bin` file to be loaded by the ConfD SNMP agent, do

confdc -c FOO-MIB.mib foo.fxs -f types.fxs

During translation, problems are reported as warnings or errors. When an error occurs, translation continues so that a complete MIB file is still produced, but the exit status from **confdc** is 1, rather than 0.

17.3.2. `--generate-oids`

With the option `--generate-oids`, the translator selects all nodes, inventing OIDs for the nodes which don't already have an `tailf:snmp-oid` statement. If a node has a `tailf:snmp-exclude-object` statement, it is ignored. The `--generate-oids` is useful when the original YANG module cannot be modified.

By default, the OID suffixes of child elements are numbered consecutively, starting with 1. This can be overridden with a suffix `tailf:snmp-oid` on a node. The suffixes of the following elements will continue from that point.

A RowStatus element is always generated.

Since the MIB compiler (i.e., **confdc -c** when given a MIB file) needs to know the association between MIB objects and YANG nodes, and this association is not present in the YANG module (if it was, there would be no need for generating OIDs), we use YANG annotation files.

A YANG annotation file is used to define the mapping between YANG nodes and MIB objects. The YANG annotation file can be written by hand, or generated from the YANG module. Since it is important in SNMP that OIDs once assigned do not change, it is recommended to generate an initial version of a YANG annotation file, and then update it manually as the YANG module evolves. The process to do this is:

- Compile the YANG module: **confdc -c foo.yang**
- Generate an initial YANG annotation file: **confdc --emit-mib FOO-MIB.mib --oid experimental.1 --generate-oids --generate-yang-annotation foo.fxs**

The YANG annotation file will be called `foo-ann.yang`.

Once the initial YANG annotation file is generated, it should be kept and updated as the original YANG module is updated. The MIB can then be generated as needed:

- Compile the YANG module and annotation file: **confdc -c -a foo-ann.yang foo.yang**
- Generate the MIB: **confdc --emit-mib FOO-MIB.mib foo.fxs**
- Compile the MIB: **confdc -c FOO-MIB.mib foo.fxs**

17.3.3. Lexical structure

At the start of the generated MIB file, a header of comments gives some information on how the file was produced, including the **confdc** invocation, the namespace of the `.fxs` file, and the current date and time. (Any `--` strings are converted to `++` because the former cannot occur in SMI comments.)

The only field in the module header which can be filled in with information from the `.fxs` file is the first `DESCRIPTION` field, which is taken from the YANG module's `description` statement, if one exists.

The remaining fields have the following format, in order to facilitate automatic editing of the values:

```
LAST-UPDATED "@LAST-UPDATED"
ORGANIZATION "@ORGANIZATION"
CONTACT-INFO "@CONTACT-INFO"
REVISION "@REVISION"
DESCRIPTION "@REVISION-DESCRIPTION"
```

17.3.4. Names

The names of the objects are constructed by joining all the parts of the full tag path of the nodes, capitalizing each part. An alternative is to not capitalize, and join the parts with "`-`" (with the option `--join-names hyphen` see the section called “Emit SMIv2 MIB options”).

If the statement `tailf:snmp-name` is used on a node, its value is taken as the full name of the object. For containers and tables, it also becomes the first part of its children's names.

The characters '`.`' and '_' can occur in YANG identifiers but not in SNMP identifiers; they are converted to '`-`', unless the option `--join-names force-capitalize` is given. In this case, such identifiers are capitalized (to lowerCamelCase).

If generated names clash with each other (for example both `/x/a/b-c` and `/x/a-b/c` yielding the name `x-a-b-c`), an error is reported.

17.3.5. Types

YANG types are mapped according to the table in Table 17.2, “YANG mapping to SMI types”. .

The type restrictions that deal with lengths ranges are translated. The remaining restrictions (`pattern`, `fraction-digits`) are silently ignored.

If inet:ipv6-address is used, Ipv6Address is imported from IPV6-TC. Otherwise, imports are only made from `SNMPv2-SMI`, `SNMPv2-CONF` and `SNMPv2-TC`.

Leafs with types which are not handled are skipped in the translation, with a warning.

Identifiers which have an invalid syntax (for example, start with a digit) are kept in the translation, but a warning is given.

If a leaf with type yang:counter64 is used as an index or as writable, a warning is given.

17.3.6. Miscellaneous

STATUS is `current` by default for all objects. To cause STATUS to be `deprecated` or `obsolete`, use the YANG statement `status` with the corresponding value on the YANG node.

MAX-ACCESS is `read-only` for operational nodes (having `config false;`).

Actions are silently ignored.

Before each generated OBJECT-TYPE and OBJECT IDENTIFIER, a comment containing the word "tag-path" indicates which YANG node the object corresponds to.

`DESCRIPTION` is copied from the `.fxs` file, when available (if the `description` statement is present). For containers, they are emitted as comments instead (the string "`--`" is replaced with "`- -`"). `description` for nodes that are not translated into any OID are lost. Double quote characters, which can't occur in `DESCRIPTION`, are replaced with single quotes.

For a string with a min length, but no max length, the max length is assumed to be 65535.

`tailf:sort-order snmp-implied;` results in the `IMPLIED` keyword, if appropriate for the type.

If a type containing negative values is used as an index, or if a string with unlimited length is used as an index, a warning is given.

If a list uses `tailf:sort-order normal`, the child nodes may not appear in SNMP order when listed, and so some elements may get lost, or confuse the manager. A warning is given for such cases.

If `tailf:sort-order snmp-implied` is used for one list list, which contains another list, the last index of the outer list (with implied length) can no longer have implied length in SNMP, so agent communication will most likely fail.

`DEFVAL` clauses are emitted for string and integer types (including bit types), but not for others.

17.4. Configuring the SNMP Agent

Configuration data for the ConfD SNMP agent consists of:

- data in `confd.conf`
- data for built-in MIBs

The configuration data in `confd.conf` is typically only configured once and then never changed. (It is possible to change however).

To store initial data for the built-in MIBs, CDB init files can be used. CDB init files are loaded the first time the system is started and the database is initialized with this data. See Section 5.8, “Loading initial data into CDB”. Typically these files will define access rules and users of the agent. Updating the MIB data is done directly from the northbound agents such as NETCONF or CLI. The **confdc** compiler flag `--`

`export` can be used to grant access for applications / protocols such as NETCONF and CLI to modify the built-in SNMP data. For this reason the YANG source for the built-in MIBs are provided so that they can be recompiled with the `--export` flag.

The data for the SNMP agent built-in MIBs are by default stored in CDB, but it is also possible have this data in an external database. In this case the user needs to add external callpoints to the YANG modules and recompile them.

17.4.1. Configuration data in `confd.conf`

There are a few elements that must be configured in `confd.conf` in order for the SNMP agent to start. First of all the ConfD SNMP agent must be *enabled*, and it must have an address and a port that it will try to bind to at start-up. If if fails to bind to the port, ConfD will fail to start. It should also have a list of MIBs that should be loaded into the agent. If it fails to load any of the MIBs, ConfD will fail to start.

Several options can be given to control the behavior of the SNMP agent:

`enabled`	Whether or not the agent should be started.
`ip, port`	The IP address and port that the agent will bind and listen for incoming requests to.
`mibs`	The MIBs that the agent should load at start-up.
`snmpVersions`	The version of the SNMP protocol that the agent will understand (the supported versions are v1, v2c, and v3).
`snmpEngineID, snmpEngineMaxMessageSize`	The engine identifier and max message size that this agent can handle.
`sysDescr`	Description of the entity. The description should include the full name and version identification of the system. See SNMPv2-MIB for more information.
`sysObjectID`	The vendor's authoritative identification of the network management subsystem contained in the entity. See SNMPv2-MIB for more information.
`sysServices`	A value which identifies the set of services that this entity primarily offers. See SNMPv2-MIB for more information.
`sysORTable`	An optional table with SNMP agent capabilities. These entries will populate the real sysORTable in the SNMPv2-MIB.

Below is an example of a `confd.conf` file:

Example 17.17. Example of a confd.conf

```
<snmpAgent>
   <enabled>true</enabled>
   <ip>0.0.0.0</ip>
   <port>161</port>
   <mibs>
      <file>SIMPLE-MIB.bin</file>
   </mibs>
   <snmpVersions>
```

```
      <v1>true</v1>
      <v2c>true</v2c>
      <v3>false</v3>
   </snmpVersions>
    <snmpEngine>
      <snmpEngineID>80:00:61:81:05:01</snmpEngineID>
    </snmpEngine>
   <system>
      <sysDescr>Tail-f ConfD agent</sysDescr>
      <sysObjectID>1.3.6.1.4.1.24961</sysObjectID>
   </system>
</snmpAgent>
```

This will enable the SNMP agent, which will listen to requests incoming to locally at port 161. The MIB file `SIMPLE-MIB.bin` is loaded in the agent, and the agent will understand SNMP versions v1 and v2c, but not v3.

17.4.2. Changing configuration data in `confd.conf` in run-time

The data stored in `confd.conf` can be changed by modifying the file and then issuing a **confd --reload** command. This will trigger an already running ConfD daemon to check its configuration data and make the necessary changes. Certain changes like the SNMP agents IP address will trigger an internal restart of the SNMP agent (ConfD will still remain up), other changes like the MIBs that are loaded can be done without restarting the SNMP agent. It's thus possible to update the MIBs in a running ConfD SNMP agent without restarting the SNMP agent.

Example 17.18. Old confd.conf content

```
<snmpAgent>
  <enabled>true</enabled>
  <ip>0.0.0.0</ip>
  <port>161</port>
  <mibs>
    <file>SIMPLE-MIB.bin</file>
  </mibs>
</snmpAgent>
```

Example 17.19. Updated confd.conf content

```
<snmpAgent>
  <enabled>true</enabled>
  <ip>0.0.0.0</ip>
  <port>161</port>
  <mibs>
    <file>SIMPLE-MIB.bin</file>
    <file>SIMPLE-MIB2.bin</file>
  </mibs>
</snmpAgent>
```

The example above will on a **confd --reload** command unload all the loaded MIBs that were specified on the old configuration, and load the MIBs specified in the updated configuration. The MIB `SIMPLE-MIB.bin` is unloaded and then loaded again, which can be useful during development to update to a newer version of the MIB.

17.4.3. Built-in MIB data

There is a set of standard MIBs which are used to control and configure an SNMP agent. These MIBs are implemented in this agent. The user can control which of these MIBs are actually loaded into the agent, and thus made visible to SNMP managers. For example, in a non-secure environment, it might be a good idea to not make MIBs that define access control visible. Note, the data that the MIBs define is used internally in the agent, even if the MIBs are not loaded.

Any SNMP agent must implement the *system* group and the *snmp* group, defined in SNMPv2-MIB. This MIB will be loaded by default.

An SNMPv3 agent must implement the SNMP-FRAMEWORK-MIB and SNMP-MPD-MIB. These MIBs are also loaded by default, if the agent is configured for SNMPv3.

There are five other standard MIBs, which also may be loaded into the agent. These MIBs are:

- SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB which defines managed objects for configuration of management targets, i.e. receivers of notifications (traps and informs). These MIBs can be used with any SNMP version.
- SNMP-VIEW-BASED-ACM-MIB, which defines managed objects for access control. This MIB can be used with any SNMP version.
- SNMP-COMMUNITY-MIB, which defines managed objects for coexistence of SNMPv1 and SNMPv2c with SNMPv3. This MIB is only useful if SNMPv1 or SNMPv2c is used, possibly in combination with SNMPv3.
- SNMP-USER-BASED-SM-MIB, which defines managed objects for authentication and privacy. This MIB is only useful with SNMPv3.

Initial config data for communities, access rules, users etc. is preferably stored in CDB init files. They are read once when the system is started for the first time and put in the database. The files must be located in the *dbDir*. Typically a system have the following CDB init files for the built-in MIBs (the file name can be anything but must end with the suffix `_init.xml`):

`community_init.xml`	Data for SNMP-COMMUNITY-MIB. Defines the communities that have access to the system.
`vacm_init.xml`	Data for SNMP-VIEW-BASED-ACM-MIB. Defines view based access.
`usm_init.xml`	Data for SNMP-USER-BASED-SM-MIB. Defines user based access used with authentication and privacy. This is only used with SNMP v3.
`target_init.xml`	Data for SNMP-TARGET-MIB. Defines trap targets.
`notify_init.xml`	Data for SNMP-NOTIFICATION-MIB. Defines notifications.

Below is an example of an init file to define a community within the agent.

Example 17.20. Example community_init.xml

```
<SNMP-COMMUNITY-MIB
        xmlns="http://tail-f.com/ns/mibs/SNMP-COMMUNITY-MIB/200308060000Z">
  <snmpCommunityTable>
    <snmpCommunityEntry>
```

```
      <snmpCommunityIndex>public</snmpCommunityIndex>
      <snmpCommunityName>public</snmpCommunityName>
      <snmpCommunitySecurityName>public</snmpCommunitySecurityName>
      <snmpCommunityContextEngineID>80:00:61:81:05:01</snmpCommunityContextEngineID>
      <snmpCommunityContextName/>
      <snmpCommunityTransportTag/>
      <snmpCommunityStorageType>permanent</snmpCommunityStorageType>
    </snmpCommunityEntry
  </snmpCommunityTable>
</SNMP-COMMUNITY-MIB>
```

SNMP-TARGET-MIB

The following values are supported for the object `snmpTargetAddrTDomain`:

- `SNMPv2-TM::snmpUDPDomain` UDP over IPv4
- `TRANSPORT-ADDRESS-MIB::transportDomainUdpIpv4` UDP over IPv4 (same as snmpUDPDomain above)
- `TRANSPORT-ADDRESS-MIB::transportDomainUdpIpv6` UDP over IPv6

SNMP-USER-BASED-SM-MIB

The following authentication algorithms are supported:

- `SNMP-USER-BASED-SM-MIB::usmNoAuthProtocol` No Authentication Protocol.
- `SNMP-USER-BASED-SM-MIB::usmHMACMD5AuthProtocol` The HMAC-MD5-96 Digest Authentication Protocol.
- `SNMP-USER-BASED-SM-MIB::usmHMACSHAAuthProtocol` The HMAC-SHA-96 Digest Authentication Protocol.

The following privacy algorithms are supported:

- `SNMP-USER-BASED-SM-MIB::usmNoPrivProtocol` No Privacy Protocol.
- `SNMP-USER-BASED-SM-MIB::usmDESPrivProtocol` The CBC-DES Symmetric Encryption Protocol.
- `SNMP-USM-AES-MIB::usmAesCfb128Protocol` The CFB128-AES-128 Privacy Protocol.

17.5. How the SNMP Agent Interacts with ConfD

17.5.1. ConfD Sessions and Transactions

All data access to ConfD is done through *user sessions*. Once a user session is established, it can open read-only or read-write *transactions* towards a data store or towards operational data.

All requests start transactions towards the running data store.

SNMP over UDP does not have a concept of a user session. Each packet is (in theory) handled independently. However, for performance reasons, the SNMP agent creates a user session and a transaction when

it receives the first packet. It then caches the session and transaction, and if it gets a new packet with the same source IP address, same UDP port, and same securityName, it reuses the session and transaction.

The cache has a limit of 16 sessions. If the cache is full when the agent needs to create a new session, an old session will be closed to give room to the new session.

If no packet has been received during a 10 second period, the session and transaction are closed.

The `confd.conf` parameters for session limits can be used to limit the number of concurrent SNMP user sessions or configuration transactions.

17.5.2. USM and VACM and ConfD AAA

When the SNMP agent receives a SNMP request, it determines the securityName and SNMP context for the request. If SNMPv1 or SNMPv2c is used, the snmpCommunityTable is consulted to determine the securityName and SNMP context. If SNMPv3 is used, the SNMP context is explicitly given in the request, and the securityName is determined from the usmUserTable.

When the SNMP agent starts a user session in ConfD , it uses the securityName as the username, the string "snmp" as ConfD AAA context, and no groups. If the username is a member of any of ConfD 's AAA groups, it will be placed in these groups. Otherwise, if there is a defaultGroup configured in `confd.conf` , the user will be placed in this group. Otherwise, the user does not belong to any group.

Note that the user is authenticated by the SNMP agent, and not by ConfD 's AAA.

For each SNMP object the user tries to access, VACM is consulted to see if the user's securityName has access, in the given context. If it has, the SNMP agent will try to access the corresponding YANG object. ConfD 's normal AAA authorization is consulted to see if the groups the user belongs to have access to the YANG object.

Since both VACM and ConfD 's AAA are consulted, a ConfD user can choose to use one of them, or both. One usage strategy can be to add VACM rules which gives full access to everyone, and then rely on ConfD 's AAA datarules. Another strategy could be to have detailed rules in VACM, and then give full access to the "snmp" context in ConfD 's AAA.

17.5.3. ConfD High Availability

If ConfD is run in HA mode, the SNMP variables `sysUpTime`, `snmpEngineTime`, and `snmpEngineBoots` are automatically replicated. This means that if a slave ConfD takes over as master, these variables will keep their values.

It is essential that each ConfD instance in the cluster has the same `snmpEngineID` configured. This value is defined in `confd.conf` , and it is the responsibility of the user to make sure it has the same value on all nodes in the cluster. However, if ConfD 's configuration is stored in CDB (see Section 27.4.2, “Storing ConfD configuration parameters in CDB”), then since CDB is replicated, the `snmpEngineID` will always be the same in the cluster.

17.6. Running the SNMP Agent as a NET-SNMP subagent

The ConfD integrated SNMP agent can run as subagent to the NET-SNMP master agent. This is useful in scenarios where you want to use NET-SNMP agents for monitoring the host, or reuse other NET-SNMP subagents in your solution.

The easiest way to run the agent as sub-agent is to configure the proxy alternative in NET-SNMP `snmpd.conf`. (See the snmpd.conf man page) Make sure that you have created an access view with the correct OID root. You need to add a proxy command entry to the `snmpd.conf` file.

```
proxy [-Cn CONTEXTNAME] [SNMPCMD_ARGS] HOST OID [REMOTEOID]
```

Values for the proxy configuration:

- SNMPCMD_ARGS : these are the authentication parameters you want to pass to the ConfD SNMP agent. Note that the original auth parameters will not be used. (See snmpcmd man page). In the simplest configuration you specify a community string, for example -c secret where secret will be used as community string for all requests forwarded to ConfD .
- HOST : IPv4-address[:port] : the IP address and the port of the ConfD SNMP agent. Make sure that ConfD uses a different port than the standard ports which you probably have configured for the NET-SNMP master agent.
- OID : the SNMP OBJECT-IDENTIFIER of the root of the tree managed by ConfD SNMP agent.

After adding the values in the `snmpd.conf` file, restart the snmpd service.

The example below will forward all requests for Tail-f specific objects to the ConfD agent running on localhost on port 5000 using the communit string `secret`.

```
proxy -c secret localhost:5000 1.3.1.6.4.24961
```

Chapter 18. Web UI Development

18.1. Introduction

Cisco Network Service Orchestrator (NSO) enabled by Tail-f version 6.0 is an evolution of the Cisco acquired company, Tail-f Systems' flagship product called Network Control System or NCS. Throughout the product and its documentation this product will be referred to as NCS.

Web UI development is thought to be in the hands of the customer's frontend developers - they will know best the requirements and how to fulfill those requirements in terms of esthetics, functionality and tool chain (frameworks, libraries, external data sources and services).

ConfD comes with a nortbound interface in the shape of a JSON-RPC API. This API is designed with Web UI applications in mind, and it complies with the JSON-RPC 2.0 specification [http://www.jsonrpc.org/specification], while using HTTP/S as the transport mechanism.

The JSON-RPC API contains a handful of methods with well defined input *method* and *params*, along with the output *result*.

In addition, the API also implements a Comet model, as long polling, to allow the client to subscribe to different server events and receive event notifications about those events in near real time.

You can call these from a browser via AJAX (e.g. XMLHTTPRequest, jQuery [https://jquery.com/]) or from the command line (e.g. curl [https://github.com/bagder/curl], httpie [https://github.com/jkbr/httpie]):

```
// with jQuery
$.ajax({
  type: 'post',
  url: '/jsonrpc',
  contentType: 'application/json',
  data: JSON.stringify({
  jsonrpc: '2.0',
  id: 1,
  method: 'login',
  params: {
    'user': 'joe',
    'passwd': 'SWkkasE32'
  }
  }),
  dataType: 'json'
})
.done(function(data) {
  if (data.result)
  alert(data.result);
  else
  alert(data.error.type);
  });
```

or

```
# with curl
curl \
-X POST \
```

```
     -H 'Content-Type: application/json' \
     -d '{"jsonrpc": "2.0", "id": 1, \
     "method": "login", \
     "params": {"user": "joe", \
     "passwd": "SWkkasE32"}}' \
     http://127.0.0.1:8008/jsonrpc

     # with httpie
     http POST http://127.0.0.1:8008/jsonrpc \
     jsonrpc=2.0 id:=1 \
     method=login \
     params:='{"user": "joe", "passwd": "SWkkasE32"}'
```

18.2. Example of a common flow

You can read in the JSON-RPC API chapter about all the available methods and their signatures, but here is a working example of how a common flow would look like:

- login
- create a new read transaction
- read a value
- create a new webui (read-write) transaction, in preparation for changing the value
- change a value
- commit (save) the changes
- meanwhile, subscribe to changes and receive a notification

In the release package, under ${CONFD_DIR}/var/confd/webui/example , you will find working code to run the example below.

```
     /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'lodash',
  'bluebird',
  './JsonRpc',
  './Comet'
], function(
  _,
  Promise,
  JsonRpc,
  Comet
) {
  'use strict';

  // CHANGE AT LEAST THESE
```

```
// IN ORDER TO MAKE THIS EXAMPLE WORK IN YOUR SETUP
var jsonrpcUrl = '/jsonrpc', // 'http://localhost:8008/jsonrpc';
    path = '/dhcp:dhcp/max-lease-time',
    value = Math.floor(Math.random() * 800) + 7200;

var log,
    jsonRpc,
    comet,
    funs = {},
    ths = {
      read: undefined,
      webui: undefined
    };

// UTILITY
log = function(msg) {
  document.body.innerHTML = '<pre>' + msg + '</pre>' + document.body.innerHTML;
};

// SETUP
jsonRpc = new JsonRpc({
  url: jsonrpcUrl,
  onError: function(method, params, deferred, reply) {
    var error = reply.error,
        msg = [method, params, reply.id, error.code, error.type, error.message];

    if (method === 'comet') {
      return;
    }

    window.alert('JsonRpc error: ' + msg);
  }
});

comet = new Comet({
  jsonRpc: jsonRpc,
  onError: function(reply) {
    var error = reply.error,
        msg = [reply.id, error.code, error.type, error.message];

    window.alert('Comet error: ' + msg);
  }
});

// CALLS FOR A COMMON SCENARIO
funs.login = function() {
  log('Logging in as admin:admin...');
  return jsonRpc.call('login', {
    user: 'admin',
    passwd: 'admin'
  }).done(function() {
    log('Logged in.');
  });
};

funs.getSystemSetting = function() {
  log('Getting system settings...');
  return jsonRpc.call('get_system_setting').done(function(result) {
    log(JSON.stringify(result, null, 1));
  });
```

```
};

funs.newReadTrans = function() {
  log('Create a new read-only transaction...');
  return jsonRpc.call('new_read_trans', {
    db: 'running'
  }).done(function(result) {
    ths.read = result.th;
    log('Read-only transaction with th (transaction handle) id: ' +
        result.th + '.');
  });
};

funs.newWebuiTrans = function() {
  log('Create a new webui (read-write) transaction...');
  return jsonRpc.call('new_webui_trans', {
    conf_mode: 'private',
    db: 'candidate'
  }).done(function(result) {
    ths.webui = result.th;
    log('webui (read-write) transaction with th (transaction handle) id: ' +
        result.th + '.');
  });
};

funs.getValue = function(args /*{th, path}*/) {
  log('Get value for ' + args.path + ' in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('get_value', {
    th: args.th,
    path: path
  }).done(function(result) {
    log(path + ' is now set to: ' + result.value + '.');
  });
};

funs.setValue = function(args /*{th, path, value}*/) {
  log('Set value for ' + args.path +
      ' to ' + args.value +
      ' in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('set_value', {
    th: args.th,
    path: path,
    value: args.value
  }).done(function(result) {
    log(path + ' is now set to: ' + result.value + '.');
  });
};

funs.validate = function(args /*{th}*/) {
  log('Validating changes in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('validate_commit', {
```

```
    th: args.th
  }).done(function() {
    log('Validated.');
  });
};

funs.commit = function(args /*{th}*/) {
  log('Commiting changes in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('commit', {
    th: args.th
  }).done(function() {
    log('Commited.');
  });
};

funs.subscribeChanges = function(args /*{path, handle}*/) {
  log('Subcribing to changes of ' + args.path +
      ' (with handle ' + args.handle + ')...');
  return jsonRpc.call('subscribe_changes', {
    comet_id: comet.id,
    path: args.path,
    handle: args.handle,
  }).done(function(result) {
    log('Subscribed with handle id ' + result.handle + '.');
  });
};

// RUN
Promise.resolve([
  funs.login,
  funs.getSystemSetting,
  funs.newReadTrans,
  function() {
    return funs.getValue({th: 'read', path: path});
  },
  function() {
    var handle = comet.id + '-max-lease-time';
    comet.on(handle, function(msg) {
      log('>>> Notification >>>\n' +
          JSON.stringify(msg, null, 2) +
          '\n<<< Notification <<<');
    });
    return funs.subscribeChanges({th: 'read', path: path, handle: handle});
  },
  funs.newWebuiTrans,
  function() {
    return funs.setValue({th: 'webui', path: path, value: value.toString()});
  },
  function() {
    return funs.getValue({th: 'webui', path: path});
  },
  function() {
    return funs.validate({th: 'webui'});
  },
  function() {
    return funs.commit({th: 'webui'});
  },
```

```
    function() {
      return new Promise(function(resolve) {
        log('Waiting 2.5 seconds before one last call to get_value');
        window.setTimeout(function() {
          resolve();
        }, 2500);
      });
    },
    function() {
      return funs.getValue({th: 'read', path: path});
    },
  ]).each(function(fn){
    return fn().then(function() {
      log('------------------------------------------------');
    });
  });
});


// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

18.3. Example of a JSON-RPC client

In the example above describing a common flow, a reference is made to using a JSON-RPC client to make the RPC calls.

An example implementation of a JSON-RPC client, used in the example above:

```
      /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'jquery',
  'lodash'
], function(
  $,
  _
) {
  'use strict';

  var JsonRpc;

  JsonRpc = function(params) {
    $.extend(this, {
      // API

      // Call a JsonRpc method with params
      call: undefined,
```

```
    // API (OPTIONAL)

    // Decide what to do when there is no active session
    onNoSession: undefined,
    // Decide what to do when the request errors
    onError: undefined,
    // Set an id to start using in requests
    id: 0,
    // Set another url for the JSON-RPC API
    url: '/jsonrpc',


    // INTERNAL

    makeOnCallDone: undefined,
    makeOnCallFail: undefined
  }, params || {});

  _.bindAll(this, [
    'call',
    'onNoSession',
    'onError',
    'makeOnCallDone',
    'makeOnCallFail'
  ]);
};

JsonRpc.prototype = {
  call: function(method, params, timeout) {
    var deferred = $.Deferred();

    // Easier to associate request/response logs
    // when the id is unique to each request
    this.id = this.id + 1;

    $.ajax({
      // HTTP method is always POST for the JSON-RPC API
      type: 'POST',
      // Let's show <method> rather than just "jsonrpc"
      // in the browsers' Developer Tools - Network tab - Name column
      url: this.url + '/' + method,
      // Content-Type is mandatory
      // and is always "application/json" for the JSON-RPC API
      contentType: 'application/json',
      // Optionally set a timeout for the request
      timeout: timeout,
      // Request payload
      data: JSON.stringify({
        jsonrpc: '2.0',
        id: this.id,
        method: method,
        params: params
      }),
      dataType: 'json',
      // Just in case you are doing cross domain requests
      // NOTE: make sure you are setting CORS headers similarly to
      // --
      // Access-Control-Allow-Origin: http://server1.com, http://server2.com
      // Access-Control-Allow-Credentials: true
      // Access-Control-Allow-Headers: Origin, Content-Type, Accept
```

```
      // Access-Control-Request-Method: POST
      // --
      // if you want to allow JSON-RPC calls from server1.com and server2.com
      crossDomain: true,
      xhrFields: {
        withCredentials: true
      }
    })
    // When done, or on failure,
    // call a function that has access to both
    // the request and the response information
    .done(this.makeOnCallDone(method, params, deferred))
    .fail(this.makeOnCallFail(method, params, deferred));
    return deferred.promise();
  },

  makeOnCallDone: function(method, params, deferred) {
    var me = this;

    return function(reply/*, status, xhr*/) {
      if (reply.error) {
        return me.onError(method, params, deferred, reply);
      }
      deferred.resolve(reply.result);
    };
  },

  onNoSession: function() {
    // It is common practice that when missing a session identifier
    // or when the session crashes or it times out due to inactivity
    // the user is taken back to the login page
    _.defer(function() {
      window.location.href = 'login.html';
    });
  },

  onError: function(method, params, deferred, reply) {
    if (reply.error.type === 'session.missing_sessionid' ||
        reply.error.type === 'session.invalid_sessionid') {
      this.onNoSession();
    }

    deferred.reject(reply.error);
  },

  makeOnCallFail: function(method, params, deferred) {
    return function(xhr, status, errorMessage) {
      var error;

      error = $.extend(new Error(errorMessage), {
        type: 'ajax.response.error',
        detail: JSON.stringify({method: method, params: params})
      });

      deferred.reject(error);
    };
  }
};

return JsonRpc;
```

```
});

// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

18.4. Example of a Comet client

In the example above describing a common flow, a reference is made to starting a Comet channel and subscribing to changes on a specific path.

An example implementation of a Comet client, used in the example above:

```
      /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'jquery',
  'lodash',
  './JsonRpc'
], function(
  $,
  _,
  JsonRpc
) {
  'use strict';

  var Comet;

  Comet = function(params) {
    $.extend(this, {
      // API

      // Add a callback for a notification handle
      on: undefined,
      // Remove a specific callback or all callbacks for a notification handle
      off: undefined,
      // Stop all comet notifications
      stop: undefined,

      // API (OPTIONAL)

      // Decide what to do when the comet errors
      onError: undefined,
      // Optionally set a different id for this comet channel
      id: 'main-1.' + String(Math.random()).substring(2),
      // Optionally give an existing JsonRpc client
      jsonRpc: new JsonRpc(),
      // Optionally wait 1 second in between polling the comet channel
      sleep: 1 * 1000,
```

```
    // INTERNAL

    handlers: [],
    polling: false,
    poll: undefined,
    onPollDone: undefined,
    onPollFail: undefined
  }, params || {});

  _.bindAll(this, [
    'on',
    'off',
    'stop',
    'onError',
    'poll',
    'onPollDone',
    'onPollFail'
  ]);
};

Comet.prototype = {
  on: function(handle, callback) {
    if (!callback) {
      throw new Error('Missing a callback for handle ' + handle);
    }

    // Add a handler made of handle id and a callback function
    this.handlers.push({handle: handle, callback: callback});

    // Start polling
    _.defer(this.poll);
  },

  off: function(handle, callback) {
    if (!handle) {
      throw new Error('Missing a handle');
    }

    // Remove all handlers matching the handle,
    // and optionally also the callback function
    _.remove(this.handlers, {handle: handle, callback: callback});

    // If there are no more handlers matching the handle,
    // then unsubscribe from notifications, in order to releave
    // the server and the network from redundancy
    if (!_.find(this.handlers, {handle: handle}).length) {
      this.jsonRpc.call('unsubscribe', {handle: handle});
    }
  },

  stop: function() {
    var me = this,
        deferred = $.Deferred(),
        deferreds = [];

    if (this.polling) {
      // Unsubcribe from all known notifications, in order to releave
      // the server and the network from redundancy
      _.each(this.handlers, function(handler) {
        deferreds.push(me.jsonRpc('unsubscribe', {
```

```
          handle: handler.handle
        }));
      });

      $.when.apply($, deferreds).done(function() {
        deferred.resolve();
      }).fail(function(err) {
        deferred.reject(err);
      }).always(function() {
        me.polling = false;
        me.handlers = [];
      });
    } else {
      deferred.resolve();
    }

    return deferred.promise();
  },

  poll: function() {
    var me = this;

    if (this.polling) {
      return;
    }

    this.polling = true;

    this.jsonRpc.call('comet', {
      comet_id: this.id
    }).done(function(notifications) {
      me.onPollDone(notifications);
    }).fail(function(err) {
      me.onPollFail(err);
    }).always(function() {
      me.polling = false;
    });
  },

  onPollDone: function(notifications) {
    var me = this;

    // Polling has stopped meanwhile
    if (!this.polling) {
      return;
    }

    _.each(notifications, function(notification) {
      var handle = notification.handle,
          message = notification.message,
          handlers = _.where(me.handlers, {handle: handle});

      // If we received a notification that we cannot handle,
      // then unsubcribe from it, in order to releave
      // the server and the network from redundancy
      if (!handlers.length) {
        return this.jsonRpc.call('unsubscribe', {handle: handle});
      }

      _.each(handlers, function(handler) {
```

```
          _.defer(function() {handler.callback(message);});
        });
      });

      _.defer(this.poll);
    },

    onPollFail: function(error) {
      switch (error.type) {
        case 'comet.duplicated_channel':
        this.onError(error);
        break;

        default:
        this.onError(error);
        _.wait(this.poll, this.sleep);
      }
    },

    onError: function(reply) {
      var error = reply.error,
          msg = [reply.id, error.code, error.type, error.message].join(' ');

      console.error('Comet error: ' + msg);
    }
  };

  return Comet;
});

// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

Chapter 19. The JSON-RPC API

19.1. JSON-RPC

19.1.1. Protocol overview

The JSON-RPC 2.0 Specification [http://www.jsonrpc.org/specification] contains all the details you need in order to understand the protocol but here is the short version.

A request payload typically looks like this:

```
{"jsonrpc": "2.0",
 "id": 1,
 "method": "subtract",
 "params": [42, 23]}
```

where the *method* and *params* properties are as defined in this manual page.

A response payload typically looks like this:

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": 19}
```

or

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32601,
   "type": "rpc.request.method.not_found",
   "message": "Method not found"}}
```

The request *id* param is returned as-is in the response to make it easy to pair requests and responses.

The batch JSON-RPC standard is dependent on matching requests and responses by *id*, since the server processes requests in any order it sees fit e.g.:

```
[{"jsonrpc": "2.0",
  "id": 1,
  "method": "subtract",
  "params": [42, 23]}
,{"jsonrpc": "2.0",
  "id": 2,
  "method": "add",
  "params": [42, 23]}]
```

with a possible response like (first result for "add", second result for "substract"):

```
[{"jsonrpc": "2.0",
  "id": 2,
  "result": 65}
,{"jsonrpc": "2.0",
  "id": 1,
  "result": 19}]
```

19.1.2. Common concepts

The URL for the JSON-RPC API is `/jsonrpc`. For logging and debugging purposes, you can add anything as a subpath to the URL, for example turning the URL into `/jsonrpc/<method>` which will allow you to see the exact method in different browsers' *Developer Tools* - *Network* tab - *Name* column, rather than just an opaque "jsonrpc".

For brevity, in the upcoming descriptions of each methods, only the input *params* and the output *result* are mentioned, although they are part of a fully formed JSON-RPC payload.

Authorization is based on HTTP cookies. The response to a successful call to *login* would create a session, and set a HTTP-only cookie, and even a HTTP-only secure cookie over HTTPS, named *sessionid*. All subsequent calls are authorized by the presence and the validity of this cookie.

The *th* param is a transaction handle identifier as returned from a call to *new_read_trans* or *new_write_trans*.

The *comet_id* param is a unique id (decided by the client) which must be given first in a call to the *comet* method, and then to upcoming calls which trigger comet notifications.

The *handle* param needs to a semantic value (not just a counter) prefixed with the comet id (for disambiguation), and overrides the handle that would have otherwise been returned by the call. This gives more freedom to the client and set semantic handles.

Common errors

The JSON-RPC specification defines the following error *code* values:

- -32700 - Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.
- -32600 - The JSON sent is not a valid Request object.
- -32601 - The method does not exist / is not available.
- -32602 - Invalid method parameter(s).
- -32603 - Internal JSON-RPC error.
- -32000 to -32099 - Reserved for application defined errors (see below)

To make server errors easier to read, along the numeric *code*, we use a *type* param that yields a literal error token. For all application defined errors, the *code* is always -32000. It's best to ignore the *code* and just use the *type* param.

```
{"jsonrpc": "2.0",
 "id": 1,
 "method": "login",
 "params":
 {"foo": "joe",
  "bar": "SWkkasE32"}}
```

which results in:

```
{"jsonrpc": "2.0",
 "id": 1,
```

```
 "error":
 {"code": -32602,
  "type": "rpc.method.unexpected_params",
  "message": "Unexpected params",
  "data":
  {"param": "foo"}}}
```

The *message* param is a free text string in English meant for human consumption, which is a one-to-one match with the *type* param. To remove noise from the examples, this param is omitted from the following descriptions.

An additional method-specific *data* param may be added to give further details on the error, most predominantly a *reason* param which is also a free text string in English meant for human consumption. To remove noise from the examples, this param is omitted from the following descriptions. But any additional *data* params will be noted by each method description.

Application defined errors

All methods may return one of the following JSON RPC or application defined errors, in addition to others, specific to each method.

```
{"type": "rpc.request.parse_error"}
{"type": "rpc.request.invalid"}
{"type": "rpc.method.not_found"}
{"type": "rpc.method.invalid_params", "data": {"param": <string>}}
{"type": "rpc.internal_error"}


{"type": "rpc.request.eof_parse_error"}
{"type": "rpc.request.multipart_broken"}
{"type": "rpc.request.too_big"}
{"type": "rpc.request.method_denied"}


{"type": "rpc.method.unexpected_params", "data": {"param": <string>}}
{"type": "rpc.method.invalid_params_type", "data": {"param": <string>}}
{"type": "rpc.method.missing_params", "data": {"param": <string>}}
{"type": "rpc.method.unknown_params_value", "data": {"param": <string>}}


{"type": "rpc.method.failed"}
{"type": "rpc.method.denied"}
{"type": "rpc.method.timeout"}

{"type": "session.missing_sessionid"}
{"type": "session.invalid_sessionid"}
```

19.1.3. FAQ

What are the security characteristics of the JSON-RPC api?

JSON-RPC runs on top the embedded web server (see "The web server" chapter), which accepts HTTP and/or HTTPS.

The JSON-RPC session ties the client and the server via an HTTP cookie, named "sessionid" which contains a randomly server-generated number. This cookie is not only secure (when the requests come over HTTPS), meaning that HTTPS cookies do not leak over HTTP, but even more importantly this cookie is

also http-only, meaning that only the server and the browser (e.g. not the JavaScript code) have access to the cookie. Furthermore, this cookie is a session cookie, meaning that a browser restart would delete the cookie altogether.

The JSON-RPC session lives as long as the user does not request to logout, as long as the user is active within a 30 minute (default value, which is configurable) time frame, as long as there are no severe server crashes. When the session dies, the server will reply with the intention to delete any "sessionid" cookies stored in the browser (to prevent any leaks).

When used in a browser, the JSON-RPC API does not accept cross-domain requests by default, but can be configured to do so via the custom headers functionality in the embedded web server, or by adding a reverse-proxy (see "The web server" chapter).

What is the proper way to use the JSON-RPC api in a cors setup?

The embedded server allows for custom headers to be se, in this case CORS headers, like:

```
Access-Control-Allow-Origin: http://webpage.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Origin, Content-Type, Accept
Access-Control-Request-Method: POST
```

A server hosted at http://server.com responding with these headers, would mean that the JSON-RPC API can be contacted from a browser which is showing a web page from http://webpage.com, and will allow the browser to make POST requests, with a limited amount of headers and with credentials (i.e. cookies).

This is not enough though, because the browser also needs to be told that your JavaScript code really wants to make a CORS request. A jQuery example would show like this:

```
// with jQuery
$.ajax({
  type: 'post',
  url: 'http://server.com/jsonrpc',
  contentType: 'application/json',
  data: JSON.stringify({
    jsonrpc: '2.0',
    id: 1,
    method: 'login',
    params: {
      'user': 'joe',
      'passwd': 'SWkkasE32'
    }
  }),
  dataType: 'json',
  crossDomain: true,       // CORS specific
  xhrFields: {             // CORS specific
    withCredentials: true  // CORS specific
  }                        // CORS specific
})
```

Without this setup, you will notice that the browser will not send the "sessionid" cookie on post-login JSON-RPC calls.

What is a tag/keypath?

A *tagpath* is a path pointing to a specific position in a YANG module's schema.

A *keypath* is a path pointing to specific position in a YANG module's instance.

These kind of paths are used for several of the API methods (e.g. *set_value*, *get_value*, *subscribe_changes*), and could be seen as XPath path specifications in abbreviated format.

Lets look at some examples using the following YANG module as input:

```
module devices {
    namespace "http://acme.com/ns/devices";
    prefix d;

    container config {
        leaf description { type string; }
        list device {
            key "interface";
            leaf interface { type string; }
            leaf date { type string; }
        }
    }
}
```

Valid tagpaths:

- `/d:config/description`
- `/d:config/device/interface`

Valid keypaths:

- `/d:config/device{eth0}/date` - the date leaf value within a device with an *interface* key set to *eth0*

Note how the prefix is prepended to the first tag in the path. This prefix is compulsory.

Restricting access to methods

The AAA infrastructure can be used to restrict access to library functions using command rules:

```
<cmdrule xmlns="http://tail-f.com/yang/acm">
  <name>webui</name>
  <context xmlns="http://tail-f.com/yang/acm">webui</context>
  <command>::jsonrpc:: get_schema</command>
  <access-operations>read exec</access-operations>
  <action>deny</action>
</cmdrule>
```

Note how the command is prefixed with "::jsonrpc:: ". This tells the AAA engine to apply the command rule to JSON-RPC API functions.

You can read more about command rules in "The AAA infrastructure" chapter in this User Guide.

19.2. Methods - commands

Method get_cmds

Get a list of the session's batch commands

Params

```
{}
```

Result

```
{"cmds": <array of cmd>}

cmd =
 {"params": <object>,
  "comet_id": <string>,
  "handle": <string>,
  "tag": <"string">,
  "started": <boolean>,
  "stopped": <boolean; should be always false>}
```

Method init_cmd

Starts a batch command

NOTE: The batch command must be listed as a named command in confd.conf/ncs.conf or else it can not be started. Read more about named commands in the confd.conf.5/ncs.conf.5 manual page.

NOTE: the *start_cmd* method must be called to actually get the batch command to generate any messages.

NOTE: As soon as the batch command prints anything on stdout it will be sent as a message and turn up as a result to your polling call to the *comet* method.

Params

```
{"th": <number>,
 "name": <string>,
 "args": <string>,
 "emulate": <boolean, default: false>,
 "width": <integer, default: 80>,
 "height": <integer, default: 24>,
 "scroll": <integer, default: 0>,
 "comet_id": <string>,
 "handle": <string, optional>}
```

The *name* param is one on the named commands defined in confd.conf/ncs.conf.

The *args* param any extra arguments to be provided to the command expect for the ones specified in confd.conf/ncs.conf.

The *emulate* param specifies if terminal emulation should be enabled.

The *width*, *height*, *scroll* properties define the screen properties.

Result

```
{"handle": <string>}
```

A handle to the batch command is returned (equal to *handle* if provided).

Method send_cmd_data

Sends data to batch command started with *init_cmd*

Params

```
{"handle": <string>,
```

```
 "data": <string>}
```

The *handle* param is as returned from a call to *init_cmd* and the *data* param is what is to be sent to the batch command started with *init_cmd*.

Result

```
{}
```

Errors (specific)

```
{"type": "cmd.not_initialized"}
```

Method start_cmd

Signals that a batch command can start to generate output.

NOTE: This method must be called to actually start the activity initiated by calls to one of the methods *init_cmd*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method suspend_cmd

Suspends output from a batch command

NOTE: the *init_cmd* method must have been called with the *emulate* param set to true for this to work

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method resume_cmd

Resumes a batch command started with *init_cmd*

NOTE: the *init_cmd* method must have been called with the *emulate* param set to true for this to work

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method stop_cmd

Stops a batch command

NOTE: This method must be called to stop the activity started by calls to one of the methods *init_cmd*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

19.3. Methods - commands - subscribe

Method get_subscriptions

Get a list of the session's subscriptions

Params

```
{}
```

Result

```
{"subscriptions": <array of subscription>}

subscription =
 {"params": <object>,
  "comet_id": <string>,
  "handle": <string>,
  "tag": <"string">,
  "started": <boolean>,
  "stopped": <boolean; should be always false>}
```

Method subscribe_cdboper

Starts a subscriber to operational data in CDB. Changes done to configuration data will not be seen here.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>,
 "path": <string>}
```

The *path* param is a keypath restricting the subscription messages to only be about changes done under that specific keypath.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an array of changes of the same type as returned by the *changes* method. See above.

Errors (specific)

```
{"type": "db.cdb_operational_not_enabled"}
```

Method subscribe_changes

Starts a subscriber to configuration data in CDB. Changes done to operational data in CDB data will not be seen here. Furthermore, subscription messages will only be generated when a transaction is successfully committed.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>,
 "path": <string>,
 "skip_local_changes": <boolean, default: false>}
```

The *path* param is a keypath restricting the subscription messages to only be about changes done under that specific keypath.

The *skip_local_changes* param specifies if configuration changes done by the owner of the read-write transaction should generate subscription messages.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an object such as:

```
{"db": <"running" | "startup" | "candidate">,
 "user": <string>,
 "ip": <string>,
 "changes": <array>}
```

The *user* and *ip* properties are the username and ip-address of the committing user.

The *changes* param is an array of changes of the same type as returned by the *changes* method. See above.

Method subscribe_poll_leaf

Starts a polling subscriber to any type of operational and configuration data (outside of CDB as well).

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"th": <number>,
 "path": <string>,
 "interval": <number>,
 "comet_id": <string>,
 "handle": <string, optional>}
```

The *path* param is a keypath pointing to a leaf value.

The *interval* is a timeout in seconds between when to poll the value.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of is a simple string value.

Method subscribe_upgrade

Starts a subscriber to upgrade messages.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an object such as:

```
{"upgrade_state": <"wait_for_init" | "init" | "abort" | "commit">,
 "timeout": <number, only if "upgrade_state" === "wait_for_init">}
```

Method subscribe_jsonrpc_batch

Starts a subscriber to JSONRPC messages for batch requests.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method having exact same structure like a JSONRPC response:

```
{"jsonrpc":"2.0",
 "result":"admin",
 "id":1}
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32602,
  "type": "rpc.method.unexpected_params",
  "message": "Unexpected params",
  "data":
  {"param": "foo"}}}
```

Method start_subscription

Signals that a subscribe command can start to generate output.

NOTE: This method must be called to actually start the activity initiated by calls to one of the methods *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade* **with no *handle*

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Result

```
{}
```

Method unsubscribe

Stops a subscriber

NOTE: This method must be called to stop the activity started by calls to one of the methods *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Result

```
{}
```

19.4. Methods - data

Method create

Create a list entry, a presence container, or a leaf of type empty

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be created.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method delete

Deletes an existing list entry, a presence container, or an optional leaf and all its children (if any)

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be deleted.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method exists

Checks if optional data exists

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be checked for existence.

Result

```
{"exists": <boolean>}
```

Method get_case

Get the case of a choice leaf

Params

```
{"th": <number>,
 "path": <string>,
 "choice": <string>}
```

The *path* param is a keypath pointing to data that contains the choice leaf given by the *choice* param.

Result

```
{"case": <string>}
```

Method show_config

Retrieves a compact string representation of the configuration

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath to the configuration to be returned in a compact string format.

Result

```
{"config": <string>}
```

19.5. Methods - data - attrs

Method get_attrs

Get node attributes

Params

```
{"th": <number>,
 "path": <string>,
 "names": <array of string>}
```

The *path* param is a keypath pointing to the node and the *names* param is a list of attribute names that you want to retrieve.

Result

```
{"attrs": <object of attribute name/value>}
```

Method set_attrs

Set node attributes

Params

```
{"th": <number>,
 "path": <string>,
 "attrs": <object of attribute name/value>}
```

The *path* param is a keypath pointing to the node and the *attrs* param is an object that maps attribute names to their values.

Result

```
{}
```

19.6. Methods - data - leafs

Method get_value

Gets a leaf value

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath pointing to a value.

Result

```
{"value": <string>}
```

Example 19.1. Method get_value

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "get_value", \
         "params": {"th": 4711, \
                    "path": "/dhcp:dhcp/max-lease-time"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {"value": "7200"}
}
```

Method get_values

Get leaf values

Params

```
{"th": <number>,
 "path": <string>,
 "leafs": <array of string>}
```

The *path* param is a keypath pointing to a container. the *leafs* param is an array of children names residing under the parent container in the YANG module.

Result

```
{"values": <array of value/error>}

value  = {"value": <string>, "access": <access>}
error  = {"error": <string>, "access": <access>} |
         {"exists": true, "access": <access>} |
         {"not_found": true, "access": <access>}
access = {"read": true, write: true}
```

NOTE: The access object has no "read" and/or "write" properties if there are no read and/or access rights.

Method set_value

Sets a leaf value

Params

```
{"th": <number>,
 "path": <string>,
 "value": <string | array>}
```

The *path* param is the keypath to give a new value as specified with the *value* param.

value can be an array when the *path* is a leaf-list node.

Result

```
{}
```

Errors (specific)

```
{"type": "data.already_exists"}
{"type": "data.not_found"}
{"type": "data.not_writable"}
```

Example 19.2. Method set_value

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "set_value", \
         "params": {"th": 4711, \
                    "path": "/dhcp:dhcp/max-lease-time", \
                    "value": "4500"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}
}
```

19.7. Methods - data - leafref

Method deref

Dereferences a leaf with a leafref type

Params

```
{"th": <number>,
 "path": <string>}
```

The *path* param is a keypath pointing to a leaf with a leafref type.

Result

```
{"paths": <array of string, a keypath to a leaf>}
```

Method get_leafref_values

Gets all possible values for a leaf with a leafref type

Params

```
{"th": <number>,
```

```
 "path": <string>,
 "skip_grouping": <boolean, default: false>,
 "keys": <array of string>}
```

The *th* param is as returned from a call to *new_read_trans* or *new_write_trans*. The *path* param is a keypath pointing to a leaf with a leafref type.

The *skip_grouping* param is by default set to false and is only needed to be set to true if if a set of sibling leafref leafs points to a list instance with multiple keys *and* if *get_leafref_values* should return an array of possible leaf values instead an array of arrays with possible key value combinations.

The *keys* param is an optional array of values that should be set if a more than one leafref statement is used within action input parameters *and* if they refer to each other using `deref()` or `current()` XPath functions.

Result

```
{"values": <array of string>}
```

19.8. Methods - data - lists

Method rename_list_entry

Renames a list entry.

Params

```
{"th": <number>,
 "from_path": <string>,
 "to_keys": <array of string>}
```

The *from_path* is a keypath pointing out the list entry to be renamed.

The list entry to be renamed will, under the hood, be deleted all together and then recreated with the content from the deleted list entry copied in.

The *to_keys* param is an array with the new key values. The array must contain a full set of key values.

Result

```
{}
```

Errors (specific)

```
{"type": "data.already_exists"}
{"type": "data.not_found"}
{"type": "data.not_writable"}
```

Method move_list_entry

Moves an ordered-by user list entry relative to its siblings.

Params

```
{"th": <number>,
 "from_path": <string>,
```

```
 "to_path": <string>,
 "mode": <"first" | "last" | "before" | "after">}
```

The *from_path* is a keypath pointing out the list entry to be moved.

The list entry to be moved can either be moved to the first or the last position, i.e. if the *mode* param is set to *first* or *last* the *to_path* keypath param has no meaning.

If the *mode* param is set to *before* or *after* the *to_path* param must be specified, i.e. the list entry will be moved to the position before or after the list entry which the *to_path* keypath param points to.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method count_list_keys

Counts the number of keys in a list.

Params

```
{"th": <number>
 "path": <string>}
```

The *path* parameter is a keypath pointing to a list.

Result

```
{"count": <number>}
```

Method get_list_keys

Enumerates keys in a list.

Params

```
{"th": <number>,
 "path": <string>,
 "chunk_size": <string>,
 "lh": <number>}
```

The *th* parameter is the transaction handle.

The *path* parameter is a keypath pointing to a list. Required on first invocation - optional in following.

The *chunk_size* parameter is the number of requested keys in the result. Optional - default is 10.

The *lh* (list handle) parameter is optional (on the first invocation) but must be used in following invocations.

Result

```
{"keys": <array of array of string>,
 "total_count": <number>,
```

```
 "lh": <number>}
```

Each invocation of *get_list_keys* will return at most *chunk_size* keys. The returned *lh* must be used in following invocations to retrieve next chunk of keys. When no more keys are available the returned *lh* will be set to `-1`.

On the first invocation *lh* can either be omitted or set to `-1`.

19.9. Methods - data - query

Method query

Starts a new query attached to a transaction handle, retrieves the results, and stops the query immediately. This is a convenience method for calling *start_query*, *run_query* and *stop_query* in a one-time sequence.

This method should not be used for paginated results, as it results in performance degradation - use *start_query*, multiple *run_query*and *stop_query*instead.

Example 19.3. Method query

```
curl \
    --cookie "sessionid=sess11635875109111642;"
    -X POST
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "query", \
         "params": {"th": 1, \
                    "xpath_expr": "/dhcp:dhcp/dhcp:foo", \
                    "result_as": "keypath-value"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"current_position": 2,
  "total_number_of_results": 4,
  "number_of_results": 2,
  "number_of_elements_per_result": 2,
  "results": ["foo", "bar"]}}
```

Method start_query

Starts a new query attached to a transaction handle. On success a query handle is returned to be in subsequent calls to *run_query*.

Params

```
{"th": <number>,
 "xpath_expr": <string, optional>,
 "selection": <array of xpath expressions, optional>
 "chunk_size": <number, optional>
 "initial_offset": <number, optional>,
 "sort", <array of xpath expressions, optional>,
 "sort_order": <"ascending" | "descending", optional>,
 "include_total": <boolean, default: true>,
 "context_node": <string, xpath expression, optional>,
 "result_as": <"string" | "keypath-value", default: "string">}
```

The *xpath_expr* param is the primary XPath expression to base the query on.

A query is a way of evaluating an XPath expression and returning the results in chunks. The primary XPath expression must evaluate to a node-set, i.e. the result. For each node in the result a *selection* Xpath expression is evaluated with the result node as its context node.

Note: The terminology used here is as defined in http://en.wikipedia.org/wiki/XPath.

For example, given this YANG snippet:

```
list interface {
  key name;
  unique number;
  leaf name {
    type string;
  }
  leaf number {
    type uint32;
    mandatory true;
  }
  leaf enabled {
    type boolean;
    default true;
  }
}
```

The *xpath_expr* could be `/interface[enabled='true']` and *selection* could be `{ "name", "number" }`.

Note that the *selection* expressions must be valid XPath expressions, e.g. to figure out the name of an interface and whether its number is even or not, the expressions must look like: `{ "name", "(number mod 2) == 0" }`.

The result are then fetched using *run_query*, which returns the result on the format specified by *result_as* param.

There are two different types of result:

- *string* result is just an array with resulting strings of evaluating the *selection* XPath expressions
- `keypath-value` result is an array the keypaths or values of the node that the *selection* XPath expression evaluates to.

This means that care must be taken so that the combination of *selection* expressions and return types actually yield sensible results (for example `1 + 2` is a valid *selection* XPath expression, and would result in the string *3* when setting the result type to *string* - but it is not a node, and thus have no keypath-value.

It is possible to sort the result using the built-in XPath function `sort-by()` but it is also also possible to sort the result using expressions specified by the *sort* param. These expressions will be used to construct a temporary index which will live as long as the query is active. For example to start a query sorting first on the enabled leaf, and then on number one would call:

```
$.post("/jsonrpc", {
  jsonrpc: "2.0",
  id: 1,
  method: "start_query",
  params:  {
    th: 1,
    xpath_expr: "/interface[enabled='true']",
    selection: ["name", "number", "enabled"],
    sort: ["enabled", "number"]
  }
```

```
})
    .done(...);
```

The *context_node* param is a keypath pointing out the node to apply the query on and the *chunk_size* param specifies how many result entries to return at a time. If set to 0 a default number will be used.

The *initial_offset* param is the result entry to begin with (1 means to start from the beginning).

Result

```
{"qh": <number>}
```

A new query handler handler id to be used when calling *run_query* etc

Example 19.4. Method start_query

```
curl \
    --cookie "sessionid=sess11635875109111642;"
    -X POST
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "start_query", \
         "params": {"th": 1, \
                    "xpath_expr": "/dhcp:dhcp/dhcp:foo", \
                    "result_as": "keypath-value"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": 47}
```

Method run_query

Retrieves the result to a query (as chunks). For more details on queries please read the description of "start_query".

Params

```
{"qh": <number>}
```

The *qh* param is as returned from a call to "start_query".

Result

```
{"position": <number>,
 "total_number_of_results": <number>,
 "number_of_results": <number>,
 "chunk_size": <number>,
 "result_as": <"string" | "keypath-value">,
 "results": <array of result>}

result = <string> |
         {"keypath": <string>, "value": <string>}
```

The *position* param is the number of the first result entry in this chunk, i.e. for the first chunk it will be 1.

How many result entries there are in this chunk is indicated by the *number_of_results* param. It will be 0 for the last chunk.

The *chunk_size* and the *result_as* properties are as given in the call to *start_query*.

The *total_number_of_results* param is total number of result entries retrieved so far.

The *result* param is as described in the description of *start_query*.

Example 19.5. Method run_query

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "run_query", \
         "params": {"qh": 22}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"current_position": 2,
  "total_number_of_results": 4,
  "number_of_results": 2,
  "number_of_elements_per_result": 2,
  "results": ["foo", "bar"]}}
```

Method reset_query

Reset/rewind a running query so that it starts from the beginning again. Next call to "run_query" will then return the first chunk of result entries.

Params

```
{"qh": <number>}
```

The *qh* param is as returned from a call to *start_query*.

Result

```
{}
```

Example 19.6. Method reset_query

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "reset_query", \
         "params": {"qh": 67}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": true}
```

Method stop_query

Stops the running query identified by query handler. If a query is not explicitly closed using this call it will be cleaned up when the transaction the query is linked to ends.

Params

```
{"qh": <number>}
```

The *qh* param is as returned from a call to "start_query".

Result

```
{}
```

Example 19.7. Method stop_query

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "stop_query", \
         "params": {"qh": 67}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": true}
```

19.10. Methods - database

Method reset_candidate_db

Resets the candidate datastore

Result

```
{}
```

Method lock_db

Takes a database lock

Params

```
{"db": <"startup" | "running" | "candidate">}
```

The *db* param specifies which datastore to lock.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked", "data": {"sessions": <array of string>}}
```

The `data.sessions` param is an array of strings describing the current sessions of the locking user, e.g. an array of "admin tcp (cli from 192.245.2.3) on since 2006-12-20 14:50:30 exclusive".

Method unlock_db

Releases a database lock

Params

```
{"db": <"startup" | "running" | "candidate">}
```

The *db* param specifies which datastore to unlock.

Result

```
{}
```

Method copy_running_to_startup_db

Copies the running datastore to the startup datastore

Result

```
{}
```

19.11. Methods - general

Method comet

Listens on a comet channel, i.e. all asynchronous messages from batch commands started by calls to *start_cmd*, *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade* ends up on the comet channel.

You are expected to have a continuous long polling call to the *comet* method at any given time. As soon as the browser or server closes the socket, due to browser or server connect timeout, the *comet* method should be called again.

As soon as the *comet* method returns with values they should be dispatched and the *comet* method should be called again.

Params

```
{"comet_id": <string>}
```

Result

```
[{"handle": <number>,
  "message": <a context specific json object, see example below>},
 ...]
```

Errors (specific)

```
{"type": "comet.duplicated_channel"}
```

Example 19.8. Method comet

```
curl \
```

```
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "subscribe_changes", \
         "params": {"comet_id": "main", \
                    "path": "/dhcp:dhcp"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {"handle": "2"}}
```

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "batch_init_done", \
         "params": {"handle": "2"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

```
curl \
    -m 15 \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "comet", \
         "params": {"comet_id": "main"}}' \
    http://127.0.0.1:8008/jsonrpc
```

hangs... and finally...

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 [{"handle": "1",
   "message":
   {"db": "running",
    "changes":
    [{"keypath": "/dhcp:dhcp/default-lease-time",
      "op": "value_set",
      "value": "100"}],
    "user": "admin",
    "ip": "127.0.0.1"}}]}
```

In this case the admin user seems to have set `/dhcp:dhcp/default-lease-time` to *100*.

Method get_system_setting

Extracts system settings such as capabilities, supported datastores, etc.

Params

```
{"operation": <"capabilities" | "customizations" | "models" | "user" | "version" | "all" |
```

The *operation* param specifies which system setting to get:

- *capabilities* - the server-side settings are returned, e.g. is rollback and confirmed commit supported
- *customizations* - an array of all webui customizations
- *models* - an array of all loaded YANG modules are returned, i.e. prefix, namespace, name
- *user* - the username of the currently logged in user is returned
- *version* - the system version
- *all* - all of the above is returned.
- (DEPRECATED) *namespaces* - an object of all loaded YANG modules are returned, i.e. prefix to namespace

Result

```
{"user:" <string>,
 "models:" <array of YANG modules>,
 "version:" <string>,
 "customizations": <array of customizations>,
 "capabilities":
 {"rollback": <boolean>,
  "copy_running_to_startup": <boolean>,
  "exclusive": <boolean>,
  "confirmed_commit": <boolean>
 },
 "namespaces": <object of YANG modules prefix/namespace>}
```

The above is the result if using the *all* operation.

Method abort

Abort a JSON-RPC method by its associated id.

Params

```
{"id": <integer>}
```

The *xpath_expr* param is the XPath expression to be evaluated.

Result

```
{}
```

Method eval_XPath

Evaluates an xpath expression on the server side

Params

```
{"th": <number>,
 "xpath_expr": <string>}
```

The *xpath_expr* param is the XPath expression to be evaluated.

Result

```
{"value": <string>}
```

19.12. Methods - messages

Method send_message

Sends a message to another user in the CLI or Web UI

Params

```
{"to": <string>,
 "message": <string>}
```

The *to* param is the user name of the user to send the message to and the *message* param is the actual message.

NOTE: The username "all" will broadcast the message to all users.

Result

```
{}
```

Method subscribe_messages

Starts a subscriber to messages.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
<string>
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of these messages depend on what has happened.

When a new user has logged in:

```
{"new_user": <number, a session id to be used by "kick_user">
 "me": <boolean, is it myself?>
```

```
 "user": <string>,
 "proto": <"ssh" | "tcp" | "console" | "http" | "https" | "system">,
 "ctx": <"cli" | "webui" | "netconf">
 "ip": <string, user's ip-address>,
 "login": <string, login timestamp>}
```

When a user logs out:

```
{"del_user": <number, a session id>,
 "user": <string>}
```

When receiving a message:

```
{"sender": <string>,
 "message": <string>}
```

19.13. Methods - rollbacks

Method get_rollbacks

Lists all available rollback files

Result

```
{"rollbacks": <array of rollback>}

rollback =
 {"nr": <number>,
  "creator": <string>,
  "date": <string>,
  "via": <"system" | "cli" | "webui" | "netconf">,
  "comment": <string>,
  "label": <string>}
```

The *nr* param is a rollback number to be used in calls to *load_rollback* etc.

The *creator* and *date* properties identify the name of the user responsible for committing the configuration stored in the rollback file and when it happened.

The *via* param identifies the interface that was used to create the rollback file.

The *label* and *comment* properties is as given calling the methods *set_comment* and *set_label* on the transaction.

Method get_rollback

Gets the content of a specific rollback file. The rollback format is as defined in a curly bracket format as defined in the CLI.

Params

```
{"nr": <number>}
```

Result

```
<string, rollback file in curly bracket format>
```

Method install_rollback

Installs a specific rollback file into a new transaction and commits it. The configuration is restored to the one stored in the rollback file and no further operations are needed. It is the equivalent of creating a new private write private transaction handler with *new_write_trans*, followed by calls to the methods *load_rollback*, *validate_commit* and *commit*.

Params

```
{"nr": <number>}
```

Result

```
{}
```

Method load_rollback

Rolls back within an existing transaction, starting with the latest rollback file, down to a specified rollback file, or selecting only the specified rollback file (also known as "selective rollback").

Params

```
{"th": <number>,
 "nr": <number>,
 "path": <string>,
 "selective": <boolean, default: false>}
```

The *nr* param is a rollback number returned by *get_rollbacks*.

The *path* param is a keypath that restrict the rollback to be applied only to a subtree.

The *selective* param, false by default, can restrict the rollback process to use only the rollback specified by *nr*, rather than applying all known rollbacks files starting with the latest down to the one specified by *nr*.

Result

```
{}
```

19.14. Methods - schema

Method get_schema

Exports a JSON schema for a selected part (or all) of a specific YANG module (with optional instance data inserted)

Params

```
{"th": <number>,
 "namespace": <string, optional>,
 "path": <string, optional>,
 "levels": <number, default: -1>,
 "insert_values": <boolean, default: false>,
 "evaluate_when_entries": <boolean, default: false>}
```

One of the properties *namespace* or *path* must be specified.

A *namespace* is as specified in a YANG module.

A *path* is a tagpath/keypath pointing into a specific sub-tree of a YANG module.

The *levels* param limits the maximum depth of containers and lists from which a JSON schema should be produced (-1 means unlimited depth).

The *insert_values* param signals that instance data for leafs should be inserted into the schema. This way the need for explicit forthcoming calls to *get_elem* are avoided.

The *evaluate_when_entries* param signals that schema entries should be included in the schema even though their "when" or "tailf:display-when" statements evaluate to false, i.e. instead a boolean *evaluated_when_entry* param is added to these schema entries.

Result

```
{"meta":
 {"namespace": <string, optional>,
  "keypath": <string, optional>,
  "prefix": <string>,
  "types": <array of type>},
 "data": <array of child>}

type = <array of {<string, type name with prefix>: <type_stack>}>

type_stack = <array of type_stack_entry>

type_stack_entry =
 {"bits": <array of string>, "size": <32 | 64>} |
 {"leaf_type": <type_stack>, "list_type": <type_stack>} |
 {"union": <array of type_stack>} |
 {"name": <primitive_type | "user_defined">,
  "info": <string, optional>,
  "readonly": <boolean, optional>,
  "facets": <array of facet, only if not primitive type>}

primitive_type =
 "empty" |
 "binary" |
 "bits" |
 "date-and-time" |
 "instance-identifier" |
 "int64" |
 "int32" |
 "int16" |
 "uint64" |
 "uint32" |
 "uint16" |
 "uint8" |
 "ip-prefix" |
 "ipv4-prefix" |
 "ipv6-prefix" |
 "ip-address-and-prefix-length" |
 "ipv4-address-and-prefix-length" |
 "ipv6-address-and-prefix-length" |
 "hex-string" |
 "dotted-quad" |
 "ip-address" |
 "ipv4-address" |
```

```
 "ipv6-address" |
 "gauge32" |
 "counter32" |
 "counter64" |
 "object-identifier"

facet_entry =
 {"enumeration": {"label": <string>, "info": <string, optional>}} |
 {"fraction-digits": {"value": <number>}} |
 {"length": {"value": <number>}} |
 {"max-length": {"value": <number>}} |
 {"min-length": {"value": <number>}} |
 {"leaf-list": <boolean>} |
 {"max-inclusive": {"value": <number>}} |
 {"max-length": {"value": <number>}} |
 {"range": {"value": <array of range_entry>}} |
 {"min-exclusive": {"value": <number>}} |
 {"min-inclusive": {"value": <number>}} |
 {"min-length": {"value": <number>}} |
 {"pattern": {"value": <string, regular expression>}} |
 {"total-digits": {"value": <number>}}

range_entry =
 "min" |
 "max" |
 <number> |
 [<number, min value>, <number, max value>]

child =
 {"kind": <kind>,
  "name": <string>,
  "qname": <string, same as "name" but with prefix prepended>,
  "info": <string>,
  "namespace": <string>,
  "is_action_input": <boolean>,
  "is_action_output": <boolean>,
  "is_cli_preformatted": <boolean>,
  "presence": <boolean>,
  "ordered_by": <boolean>,
  "is_config_false_callpoint": <boolean>,
  "key": <boolean>,
  "exists": <boolean>,
  "value": <string | number | boolean>,
  "unique_children": <array of string>,
  "is_leafref": <boolean>,
  "hidden": <boolean>,
  "default_ref":
  {"namespace": <string>,
   "tagpath": <string>
  },
  "access":
  {"create": <boolean>,
   "update": <boolean>,
   "delete": <boolean>,
   "execute": <boolean>
  },
  "config": <boolean>,
  "readonly": <boolean>,
  "suppress_echo": <boolean>,
  "type":
```

```
  {"name": <primitive_type>,
   "primitive": <boolean>
  }
  "generated_name": <string>,
  "units": <string>,
  "leafref_groups": <array of string>,
  "active": <string, active case, only if "kind" is "choice">,
  "cases": <array of case, only of "kind" is "choice">,
  "default": <string | number | boolean>,
  "mandatory": <boolean>,
  "children": <children>
 }

kind =
 "module" |
 "access-denies" |
 "list-entry" |
 "choice" |
 "key" |
 "leaf-list" |
 "action" |
 "container" |
 "leaf" |
 "list" |
 "notification"

case_entry =
 {"kind": "case",
  "name": <string>,
  "children": <array of child>
 }
```

This is a fairly complex piece of JSON but it essentially maps what is seen in a YANG module. Keep that in mind when scrutinizing the above.

The *meta* param contains meta-information about the YANG module such as namespace and prefix but it also contains type stack information for each type used in the YANG module represented in the *data* param. Together with the *meta* param, the *data* param constitutes a complete YANG module in JSON format.

Example 19.9. Method get_schema

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "get_schema", \
         "params": {"th": 2, \
                    "path": "/aaa:aaa/authentication/users/user{admin}", \
                    "levels": -1, \
                    "insert_values": true}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"meta":
  {"namespace": "http://tail-f.com/ns/aaa/1.1",
   "keypath": "/aaa:aaa/authentication/users/user{admin}",
```

```
   "prefix": "aaa",
   "types":
   {"http://tail-f.com/ns/aaa/1.1:passwdStr":
    [{"name": "http://tail-f.com/ns/aaa/1.1:passwdStr"},
     {"name": "MD5DigestString"}]}}},
 "data":
 {"kind": "list-entry",
  "name": "user",
  "qname": "aaa:user",
  "access":
  {"create": true,
   "update": true,
   "delete": true},
  "children":
  [{"kind": "key",
    "name": "name",
    "qname": "aaa:name",
    "info": {"string": "Login name of the user"},
    "mandatory": true,
    "access": {"update": true},
    "type": {"name": "string", "primitive": true}},
   ...]}}
```

Method hide_schema

Hides data which has been adorned with a "hidden" statement in YANG modules. "hidden" statements is an extension defined in the tail-common YANG module (http://tail-f.com/yang/common).

Params

```
{"th": <number>,
 "group_name": <string>,
 "passwd": <string>}
```

The *group_name* param is as defined by a "hidden" statement in a YANG module.

The *passwd* param is a password needed to hide the data that has been adorned with a "hidden" statement. The password is as defined in the confd.conf/ncs.conf file.

Result

```
{}
```

Method unhide_schema

Unhides data which has been adorned with a "hidden" statement in YANG modules. "hidden" statements is an extension defined in the tail-common YANG module (http://tail-f.com/yang/common).

Params

```
{"th": <number>,
 "group_name": <string>,
 "passwd": <string>}
```

The *group_name* param is as defined by a "hidden" statement in a YANG module.

The *passwd* param is a password needed to hide the data that has been adorned with a "hidden" statement. The password is as defined in the confd.conf/ncs.conf file.

Result

```
{}
```

Method run_action

Invokes an action or rpc defined in a YANG module.

Params

```
{"th": <number>,
 "path": <string>,
 "params": <json, optional>
 "format": <"normal" | "bracket", default: "normal">,
 "comet_id": <string, optional>,
 "handle": <string, optional>}
```

Actions are as specified in th YANG module, i.e. having a specific name and a well defined set of parameters and result. the *path* param is a keypath pointing to an action or rpc in and the *params* param is a JSON object with action parameters.

The *format* param defines if the result should be an array of key values or a pre-formatted string on bracket format as seen in the CLI. The result is also as specified by the YANG module.

Both a *comet_id* and *handle* need to be provided in order to receive notifications.

Result

```
<string | array of result>

result = {"name": <string>, "value": <string>}
```

Errors (specific)

```
{"type": "action.invalid_result", "data": {"path": <string, path to invalid result>}}
```

Example 19.10. Method run_action

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", id: 1, \
         "method": "run_action", \
         "params": {"th": 2, \
                    "path": "/dhcp:dhcp/set-clock", \
                    "params": {"clockSettings": "2014-02-11T14:20:53.460%2B01:00"}}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": [{"name":"systemClock", "value":"0000-00-00T03:00:00+00:00"},
            {"name":"inlineContainer/bar", "value":"false"},
            {"name":"hardwareClock","value":"0000-00-00T04:00:00+00:00"}]}
```

```
curl \
    -s \
```

```
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d'{"jsonrpc": "2.0", "id": 1, \
        "method": "run_action", \
        "params": {"th": 2, \
                   "path": "/dhcp:dhcp/set-clock", \
                   "params": {"clockSettings": \
    "2014-02-11T14:20:53.460%2B01:00"}, \
                   "format": "bracket"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": "systemClock 0000-00-00T03:00:00+00:00\ninlineContainer  {\n    \
     bar false\n}\nhardwareClock 0000-00-00T04:00:00+00:00\n"}
```

19.15. Methods - session

Method login

Creates a user session and sets a browser cookie

Params

```
{"user": <string>, "passwd": <string>}
```

The *user* and *passwd* are the credentials to be used in order to create a user session. The common AAA engine in ConfD/NCS is used to verify the credentials.

Result

```
{}
```

NOTE The response will have a `Set-Cookie` HTTP header with a *sessionid* cookie which will be your authentication token for upcoming JSON-RPC requests.

Example 19.11. Method login

```
curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "login", \
         "params": {"user": "joe", \
                    "passwd": "SWkkasE32"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32000,
  "type": "rpc.method.failed",
  "message": "Method failed"}}
```

```
curl \
    -X POST \
    -H 'Content-Type: application/json' \
```

```
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "login", \
         "params": {"user": "admin", \
         "passwd": "admin"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

NOTE sessionid cookie is set at this point in your User Agent (browser). In our examples, we set the cookie explicitly in the upcoming requests for clarity.

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "get_trans"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {"trans": []}}
```

Method logout

Removes a user session and invalidates the browser cookie

The HTTP cookie identifies the user session so no input parameters are needed.

Params

None.

Result

```
{}
```

Example 19.12. Method logout

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "logout"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "logout"}' \
```

```
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32000,
  "type": "session.invalid_sessionid",
  "message": "Invalid sessionid"}}
```

Method kick_user

Kills a user session, i.e. kicking out the user

Params

```
{"user": <string | number>}
```

The *user* param is either the username of a logged in user or session id.

Result

```
{}
```

19.16. Methods - session data

Method get_session_data

Gets session data from the session store

Params

```
{"key": <string>}
```

The *key* param for which to get the stored data for. Read more about the session store in the *put_session_data* method.

Result

```
{"value": <string>}
```

Method put_session_data

Puts session data into the session store. The session store is small key-value server-side database where data can be stored under a unique key. The data may be an arbitrary object, but not a function object. The object is serialized into a JSON string and then stored on the server.

Params

```
{"key": <string>,
 "value": <string>}
```

The *key* param is the unique key for which the data in the *value* param is to be stored.

Result

```
{}
```

Method erase_session_data

Erases session data previously stored with "put_session_data".

Params

```
{"key": <string>}
```

The *key* param for which all session data will be erased. Read more about the session store in the *put_session_data* method.

Result

```
{}
```

19.17. Methods - transaction

Method get_trans

Lists all transactions

Params

None.

Result

```
{"trans": <array of transaction>}

transaction =
 {"db": <"running" | "startup" | "candidate">,
  "mode": <"read" | "read_write", default: "read">,
  "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
  "tag": <string>,
  "th": <integer>}
```

Example 19.13. Method get_trans

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "get_trans"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"trans":
  [{"db": "running",
    "th": 2}]}}
```

Method new_trans

Creates a new transaction

Params

```
{"db": <"startup" | "running" | "candidate", default: "running">,
 "mode": <"read" | "read_write", default: "read">,
 "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
 "tag": <string>}
```

The *conf_mode* param specifies which transaction semantics to use when it comes to lock and commit strategies. These three modes mimics the modes available in the CLI.

The meaning of *private*, *shared* and *exclusive* have slightly different meaning depending on how the system is configured; with a writable running, startup or candidate configuration.

private (*writable running enabled*) - Edit a private copy of the running configuration, no lock is taken.

private (*writable running disabled, startup enabled*) - Edit a private copy of the startup configuration, no lock is taken.

exclusive (*candidate enabled*) - Lock the running configuration and the candidate configuration and edit the candidate configuration.

exclusive (*candidate disabled, startup enabled*) - Lock the running configuration (if enabled) and the startup configuration and edit the startup configuration.

shared (*writable running enabled, candidate enabled*) - Edit the candidate configuration without locking it.

The *tag* param is a way to tag transactions with a keyword, so that they can be filtered out when you call the *get_trans* method.

Result

```
{"th": <number>}
```

A new transaction handler id

Errors (specific)

```
{"type": "trans.confirmed_commit_in_progress"}
{"type": "db.locked", "data": {"sessions": <array of string>}}
```

The `data.sessions` param is an array of strings describing the current sessions of the locking user, e.g. an array of "admin tcp (cli from 192.245.2.3) on since 2006-12-20 14:50:30 exclusive".

Example 19.14. Method new_trans

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
         "method": "new_trans", \
         "params": {"db": "running", \
                    "mode": "read"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
```

```
 "id": 1,
 "result": 2}
```

Method delete_trans

Deletes a transaction created by *new_trans* or *new_webui_trans*

Params

```
{"th": <number>}
```

Result

```
{}
```

Method set_trans_comment

Adds a comment to the active read-write transaction. This comment will be stored in rollback files and can be seen with a call to *get_rollbacks*.

Params

```
{"th": <number>}
```

Result

```
{}
```

Method set_trans_label

Adds a label to the active read-write transaction. This label will be stored in rollback files and can be seen with a call to *get_rollbacks*.

Params

```
{"th": <number>}
```

Result

```
{}
```

19.18. Methods - transaction - changes

Method is_trans_modified

Checks if any modifications has been done to a transaction

Params

```
{"th": <number>}
```

Result

```
{"modified": <boolean>}
```

Method get_trans_changes

Extracts modifications done to a transaction

Params

```
{"th": <number>}
```

Result

```
{"changes": <array of change>}

change =
 {"keypath": <string>,
  "op": <"created" | "deleted" | "modified" | "value_set">,
  "value": <string,>,
  "old": <string>
 }
```

The *value* param is only interesting if *op* is set to one of *created*, *modified* or *value_set*.

The *old* param is only interesting if *op* is set to *modified*.

Example 19.15. Method get_trans_changes

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1, \
        "method": "changes", \
        "params": {"th": 2}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 [{"keypath":"/dhcp:dhcp/default-lease-time",
   "op": "value_set",
   "value": "100",
   "old": ""}]}
```

Method validate_trans

Validates a transaction

Params

```
{"th": <number>}
```

Result

```
{}
```

or

```
{"warnings": <array of warning>}
```

```
warning = {"paths": <array of string>, "message": <string>}
```

Errors (specific)

```
{"type": "trans.resolve_needed", "data": {"users": <array string>}}
```

The `data.users` param is an array of conflicting usernames.

```
{"type": "trans.validation_failed", "data": {"errors": <array of error>}}

error = {"paths": <array of string>, "message": <string>}
```

The `data.errors` param points to a keypath that is invalid.

Method get_trans_conflicts

Gets the conflicts registered in a transaction

Params

```
{"th": <number>}
```

Result

```
{"conflicts:" <array of conflicts>}

conflict =
 {"keypath": <string>,
  "op": <"created" | "deleted" | "modified" | "value_set">,
  "value": <string>,
  "old": <string>}
```

The *value* param is only interesting if *op* is set to one of *created*, *modified* or *value_set*.

The *old* param is only interesting if *op* is set to *modified*.

Method resolve_trans

Tells the server that the conflicts have been resolved

Params

```
{"th": <number>}
```

Result

```
{}
```

19.19. Methods - transaction - commit changes

Method validate_commit

Validates a transaction before calling *commit*. If this method succeeds (with or without warnings) then the next operation *must* be all call to either *commit* or *clear_validate_lock*. The configuration will be locked for access by other users until one of these methods are called.

Params

```
{"th": <number>}
```

Result

```
{}
```

or

```
{"warnings": <array of warning>}

warning = {"paths": <array of string>, "message": <string>}
```

Errors (specific)

Same as for the *validate_trans* method.

Method clear_validate_lock

Releases validate lock taken by *validate_commit*

Params

```
{"th": <number>}
```

Result

```
{}
```

Method commit

Copies the configuration into the running datastore.

Params

```
{"th": <number>,
 "timeout": <number, default: 0>,
 "flags": <number>, default: 0}
```

The *flags* param can bit or:ed with the following values to change the commit behavior:

- `1 << 0` - Do not release locks
- `1 << 1` - Disable backlog
- `1 << 2` - Do not drop revision
- `1 << 3` - No device manager

The *timeout* param represents the confirmed commit timeout, i.e. set it to zero (0) to commit without timeout.

- If a call to *confirm_commit* is not done within *timeout* seconds an automatic rollback is performed. This method can also be used to "extend" a confirmed commit that is already in progress, i.e. set a new timeout or add changes.

- A call to *abort_commit* can be made to abort the confirmed commit.

NOTE: Must be preceded by a call to *validate_commit*

NOTE: The transaction handler is deallocated as a side effect of this method

Result

```
{}
```

Errors (specific)

```
{"type": "trans.confirmed_commit_in_progress"}
```

Method abort_commit

Aborts the active read-write transaction

Result

```
{}
```

Method confirm_commit

Confirms the currently pending confirmed commit

Result

```
{}
```

19.20. Methods - transaction - webui

Method get_webui_trans

Gets the webui read-write transaction

Result

```
{"trans": <array of trans>}

trans =
 {"db": <"startup" | "running" | "candidate", default: "running">,
  "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
  "th": <integer>
 }
```

Method new_webui_trans

Creates a read-write transaction that can be retrieved by 'get_webui_trans'.

Params

```
{"db": <"startup" | "running" | "candidate", default: "running">,
 "conf_mode": <"private" | "shared" | "exclusive", default: "private">}
```

See 'new_trans' for semantics of the parameters and specific errors.

Result

```
{"th": <number>}
```

A new transaction handler id

Chapter 20. The web server

20.1. Introduction

This document describes an embedded basic web server that can deliver static and Common Gateway Interface (CGI) dynamic content to a web client, commonly a browser. Due to the limitations of this web server, and/or of its configuration capabilities, a proxy server is recommended to address special requirements. An Nginx example is attached and described as a basic template for such a proxy server.

20.2. Web server capabilities

The web server can be configured through settings in confd.conf - see the manual pages of the section called “CONFIGURATION PARAMETERS” .

Here is a brief overview of what you can configure on the web server:

- "toggle web server": the web server can be turned on or off
- "toggle transport": enable HTTP and/or HTTPS, set IPs, ports, redirects, certificates, etc.
- "hostname": set the hostname of the web server and decide whether to block requests for other hostnames
- "/": set the docroot from where all static content is served
- "/login": set the docroot from where static content is served for URL paths starting with /login
- "/custom": set the docroot from where static content is served for URL paths starting with /custom
- "/cgi": toggle CGI support and set the docroot from where dynamic content is served for URL paths starting with /cgi
- "non-authenticated paths": by default all URL paths, except those needed for the login page are hidden from non-authenticated users; authentication is done by calling the JSONRPC "login" method
- "allow symlinks": allow symlinks from under the docroot
- "cache": set the cache time window for static content
- "log": several logs are available to configure in terms of file paths - an access log, a full HTTP traffic/trace log and a browser/JavaScript log
- "custom headers": set custom headers across all static and dynamic content, including requests to "/jsonrpc".

In addition to what is configurable, the web server also GZip-compresses responses automatically if the browser handles such responses, either by compressing the response on the fly, or, if requesting a static file, like "/bigfile.txt", by responding with the contents of "/bigfile.txt.gz", if there is such a file.

20.3. Proxy server example

In various scenarios, the web server has to be tweaked for performance, security or just browser-compatibility purposes. Such fine-grained tweaks are not considered with the embedded web server, and should be handled by installing a so called reverse-proxy to handle these fine requirements, while forwarding some

requests to the embedded web server, either for retrieving static or dynamic content, either for accessing the JSONRPC or REST API.

In the release package, under ${CONFD_DIR}/var/confd/webui/nginx , there is a basic configuration for setting up Nginx as a reverse proxy.

This configuration has been tested with nginx 1.6.2 and higher.

Common configurations have been set to sensible defaults for performance and security, but keep in mind that this is not intended as an out-of-the-box production configuration.

A quick way to test this is by starting a ConfD example on the default port 8008 and running "make nginx" from within the ${CONFD_DIR}/var/confd/webui folder. You should now be able to open your browser at "http://localhost:8090", and interact with the running example.

The nginx configuration file starts within the ${CONFD_DIR}/var/confd/webui /nginx/nginx.conf file. A number of configurations have been enabled for a development environment only, marked with "DEV ONLY" in order to enable easy debugging in the CLI. Similarly, some configurations are marked with "CHANGEME" to denote the most important configurations that you should adapt to your environment, if you will start using this configuration as a base for a production-quality nginx configuration.

Under the ${CONFD_DIR}/var/confd/webui /nginx/conf.d folder there are modular configurations. Some of them are not enabled by default in "nginx.conf" such as "sec.cors.conf" which enables Cross-Origin Resource Sharing on your server.

Chapter 21. The REST API

21.1. Introduction

This document describes a RESTful API over HTTP for accessing data defined in YANG. It tries to follow the RESTCONF Internet Draft [draft-ietf-netconf-restconf-00] but since it predates the creation of RESTCONF, a number of differences exists. Whenever such a difference occur, it will be clearly stated.

The RESTCONF protocol operates in the configuration datastores defined in NETCONF. It defines a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access these datastores. The YANG language defines the syntax and semantics of datastore content, operational data and protocol operations. REST operations are used to access the hierarchical data within a datastore. Request and response data can be in XML or JSON format, where XML is the default format.

To get a quick introduction to how to enable REST in ConfD and how to run the CRUD operations, continue to Section 21.2, “Getting started”.

To read more about resources, continue to Section 21.4, “Resources”.

Tip

All REST request/response examples in this chapter are based on the *examples.confd/rest/router* example. Go to this example to play around and get familiar with the REST api.

21.2. Getting started

In order to enable REST in ConfD , REST must be enabled in `confd.conf` . The web server configuration for REST is shared with the WebUI's config. However, the WebUI does not have to be enabled for REST to work.

Here's a minimal example of what is needed in the conf file:

Example 21.1. ConfD configuration for REST

```
<rest>
  <enabled>true</enabled>
</rest>

<webui>
  <enabled>false</enabled>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8008</port>
    </tcp>
  </transport>
</webui>
```

21.2.1. Request URI structure

Resources are represented with URIs following the structure for generic URIs in [RFC3986].

Example 21.2. Request URI structure

```
 <OP> /api/<path>?<query>#<fragment>

  ^    ^     ^        ^       ^
  |    |     |        |       |
method entry resource query fragment

  M      M      O       O     I

M=mandatory, O=optional, I=ignored


<text> replaced by client with real values
```

A REST operation is derived from the HTTP method and the request URI, using the following conceptual fields:

- "method": the HTTP method identifying the REST operation requested by the client, to act upon the target resource specified in the request URI.
- "entry": the well-known REST entry point ("/api").

  Note
  THIS DIFFERS FROM RESTCONF!

- "resource": the path expression identifying the resource that is being accessed by the operation. If this field is not present, then the target resource is the API itself, represented by the media type "application/vnd.yang.api".

  Note
  THE MEDIA TYPE DIFFERS FROM RESTCONF!

- "query": the set of parameters associated with the REST message. These have the familiar form of "name=value" pairs. There is a specific set of parameters defined, see Section 21.2.10, “Query Parameters”. The contents of the query parameter value must be encoded according to [RFC2396], section 3.4. Any reserved characters must be encoded with escape sequences, according to [RFC2396], section 2.4.
- "fragment": This field is not used by the REST protocol.

The REST protocol uses HTTP methods to identify the CRUD operation requested for a particular resource. The following table shows how the REST operations relate to NETCONF protocol operations:

Table 21.1. REST vs NETCONF operations

REST	NETCONF
GET	<get-config>, <get>
POST	<edit-config> (operation="create")
PUT	<edit-config> (operation="replace")
PATCH	<edit-config> (operation="merge")
DELETE	<edit-config> (operation="delete")
OPTIONS	none

REST	NETCONF
HEAD	none

21.2.2. Accessing the REST API

The REST API can be accessed, e.g., by using **curl**:

Example 21.3. Using curl for accessing ConfD

```
curl -u admin:admin -s http://localhost:8008/api/foo/bar -X GET
```

To provide an HTTP header use the `-H` switch:

```
... -H "Accept: application/vnd.data+xml"
```

Note that in the following examples we will shorten the curl calls to:

```
GET /foo/bar
Accept: application/vnd.data+xml
```

By default curl will display responses on standard output. The headers are not included. To include these the "-i" switch has to be added.

```
curl -i ...
```

The response can then typically look like:

```
HTTP/1.1 200 OK
Server: ConfD/5.3.0
Cache-control: private, no-cache, must-revalidate, proxy-revalidate
Date: Fri, 05 Sep 2014 13:09:46 GMT
Content-Type: application/vnd.yang.data+json
Transfer-Encoding: chunked
Etag: 1409-922585-953711
Last-Modified: Fri, 01 Jan 1971 00:00:00 GMT

{
  "some-data": {
    "some-value": "value"
  }
}
```

In the examples shown in this chapter we will from the beginning show all headers with the response. In later examples the headers will be omitted for brevity.

21.2.3. GET

The GET method is sent by the client to retrieve data and meta-data for a resource. It is supported for all resource types, except operation resources. The request must contain a request URI that contains at least the entry point component (`/api`). Note how we make use of the `Accept` HTTP header to indicate what format we want the returned result to be in. The value of the `Accept` HTTP header, in this example: `application/yang.data+json`, must be a valid media type. Since XML is the default format we need to explicitly request JSON in the example below. To read more about the media types, see Section 21.4, “Resources”.

Example 21.4. Get the "sys/interfaces" resource represented as JSON

The server might respond:

> **Note**
> To indicate a particular YANG namespace in the URI, the YANG module prefix is used. In the example we are using a module prefix: `ex` .
> *THIS DIFFERS FROM RESTCONF!*

In accordance with RESTCONF, the returned `"serial"` container is using a module name: `example-serial` to indicate a particular YANG namespace.

To read more about the `ETag` and `Last-Modified` response headers, see Section 21.6, “Request/Response headers”.

Refer to Section 21.3.1, “GET and Query examples” for more resource retrieval examples.

21.2.4. POST

The POST method is sent by the client to create a data resource or invoke an operation resource ("rpc" or "tailf:action").

Here we show an example of resource creation using POST. For an example of operation invokation see Section 21.3.3, “Invoke Operations”

Example 21.5. Create a new "sys/routes/inet/route" resource, with JSON payload

```
{
  "route": {
    "name": "10.20.1.0",
    "prefix-length": "24"
  }
}
```

If the resource is created, the server might respond as follows:

If the POST method succeeds, a `"201 Created"` Status-Line is returned and there is no response message body. Also, a `"Location"` header identifying the child resource that was created is present in the response.

Refer to the section called “Create a List Instance with POST” for more examples of creating resources.

If the target resource type is an operation resource, then the POST method is treated as a request to invoke that operation. The message body (if any) is processed as the operation input parameters. Refer to Section 21.4.5, “Operations and Actions” for details on operation resources.

21.2.5. PUT

The PUT method is sent by the client to create or replace the target resource. The request must contain a request URI that contains a target resource that identifies the data resource to create or replace.

Example 21.6. Replace the "sys/routes/inet/route" resource contents

```
{
  "route": {
    "name": "10.20.1.0",
    "prefix-length": "24",
    "description": "Example route",
    "enabled" : "false"
  }
}
```

If the resource is updated, the server might respond:

The "insert" and "resource" query parameters are supported by the PUT method for data resources, for more examples see the section called “Insert Data into Resources”.

> **Note**
> The "resource" query parameter correspond to the "point" query parameter in RESTCONF.
> *THIS DIFFERS FROM RESTCONF!*

If the PUT method creates a new resource, a "201 Created" Status-Line is returned. If an existing resource is modified, either "200 OK" or "204 No Content" are returned.

Refer to the section called “Create and Replace a List Instance with PUT” for more examples on how to use PUT.

21.2.6. PATCH

The PATCH method is used to create or update a sub-resource within the target resource. If the target resource instance does not exist, the server *WILL NOT* create it.

To replace just the "enabled" field in the "route" list resource (instead of replacing the entire resource with the PUT method), the client might send a plain patch as follows.

Example 21.7. Update the "sys/routes/inet/route" resource contents

```
{
  "route": {
    "enabled" : "true"
  }
}
```

If the resource is updated, the server might respond:

Refer to the section called “Update Existing List Instance with PATCH” for more examples on how to use PATCH.

21.2.7. DELETE

The DELETE method is used to delete the target resource. If the DELETE method succeeds, a "204 No Content" Status-Line is returned, and there is no response message body.

Example 21.8. Delete the "sys/routes/inet/route" resource contents

If the resource is successfully deleted, the server might respond:

Refer to Section 21.3.4, “Delete Data Resources” for more examples on how to use DELETE.

21.2.8. OPTIONS

The OPTIONS method is sent by the client to discover which methods are supported by the server for a specific resource. The supported methods are listed in the ALLOW header.

Example 21.9. Get options for the "sys" resource

The server might respond:

Here the options method responds with an ALLOW header indicating that all methods are allowed on the "sys" resource.

21.2.9. HEAD

The HEAD method is sent by the client to retrieve just the headers that would be returned for the comparable GET method, without the response body. The access control behavior is enforced as if the method was GET instead of HEAD. The server will respond the same as if the method was GET instead of HEAD, except that no response body is included.

Example 21.10. Get head for the "sys/interfaces/ex:serial" resource

The server might respond:

Here the "Content-Length" header is 0. This is because the content length calculation is not eligible when the actual content is suppressed as with the HEAD method.

21.2.10. Query Parameters

Each REST operation allows zero or more query parameters to be present in the request URI. The specific parameters that are allowed depends on the resource type, and sometimes the specific target resource used, in the request.

Table 21.2. Query Parameters

Name	Methods	Description
deep	GET	Retrieve a resource with all subresources inline.
insert	POST	For an ordered-by user list, we can specify where a resource, to be created, should be inserted. This query parameter is used together with the *resource* query parameter. Possible values are: *after, before,first* and *last*. See the section called “Insert Data into Resources” for details.
limit	GET	Used by the client to specify a limited set of list entries to retrieve. See the section called “Partial Responses” for details.
offset	GET	Used by the client to specify a limited set of list entries to retrieve. See the section called “Partial Responses” for details.
operations	GET	Used by the client to include/exclude operations (tailf:actions) in the result. Possible values are: *true*, which is the default value, and *false*.
resource	POST	For an ordered-by user list, we can specify where a resource, to be created, should be inserted. This query parameter is used together with the *insert* query parameter. See the section called “Insert Data into Resources” for details.
select	GET	Used by the client to select which nodes and subresources in a resource to retrieve. See the section called “Partial Responses” for details.
shallow	GET	Retrieve a resource with no subresources inline.
unhide	GET	Used by the client to unhide hidden nodes. See the section called “Hidden Nodes” for details.
verbose	GET	Used by the client to control display of the "self" and "path" attributes of the resource. See the section called “Displaying Default Data” for details.
with-defaults	GET	Used by the client to control display of default data in GET requests. See the section called “Displaying Default Data” for details.

Note
The query parameters of the REST API is not the same as for RESTCONF.
THIS DIFFERS FROM RESTCONF!

If neither "deep" nor "shallow" is used, you will get a variable depth returned. The output shown will stop at the first encountered presence container or list key value(s).

21.3. Resource Examples

21.3.1. GET and Query examples

Retrieve Data Resources

If a resource only needs to be outlined, the *shallow* query parameter can be used on the GET request.

Example 21.11. Shallow get for the "sys" resource

The server might respond:

On the other hand, if the full subtree under a resource is required, the *deep* query parameter can be used on the GET request.

Example 21.12. Deep get for the "sys/interfaces/interface" resource

The server might respond:

For more info about "deep" vs "shallow", see: Section 21.2.10, “Query Parameters” [407]

Partial Responses

By default, the server sends back the full representation of a resource after processing a request. For better performance, the server can be instructed to send only the nodes the client really needs in a partial response.

To request a partial response for a set of list entries, use the "offset" and "limit" query parameters to specify a limited set of entries to be returned.

For example, if we what to retrieve only 2 entries from the `sys/routes/inte/route` list we can issue the command:

Example 21.13. Limit the response

The following request retrieves 2 entries starting from entry 3 (the first entry is 0):

To request a filtered partial response, use the "select" query parameter to specify the nodes and subresources to be returned.

- Use a semicolon-separated list to select multiple nodes.
- Use "a/b" to select a node "b" that is nested within node "a"; use "a/b/c" to select a node "c" nested within "b".
- Specify node subselectors to request only specific subnodes by placing expressions in parentheses "( )" after any selected node.
- To specify all subnodes of a specific node use the special wildcard notion within parentheses "(*)"

NOTE: "a/b/c;a/b/d" is equivalent to "a/b(c;d)"

The following request selects the routes name and next-hop/name nodes:

Example 21.14. Limit the response with select

The server might respond:

Hidden Nodes

Hidden nodes are described in Section 10.7, “Hidden Data”. By default, hidden nodes are not visible in the REST interface. In order to unhide hidden nodes for retrieval or editing, clients can use the query parameter "unhide". The format of the "unhide" parameter is a comma-separated list of

```
<groupname>[;<password>]
```

As an example:

```
unhide=extra,debug;secret
```

This example unhides the normal group "extra" and the password-protected group "debug" with the password "secret".

Displaying Default Data

Normally, leaf nodes that are not set but has a default value are not displayed at a GET request. This behavior can be controlled by the use of the "with-defaults" query parameter. This parameter can take one of four values:

- "report-all": all data nodes are be reported, including any data nodes considered to be default data by the server.
- "explicit": a data node that has been explicitly set is reported. This is also the case when a data node is explicitly set to a value that coincide with the default value for the node.

- "trim": data nodes are not reported if they contain default value even if this was explicitly set to that value.
- "report-all-tagged": In this mode the server returns all data nodes, just like the "report-all" mode, except a data node that is considered by the server to contain default data will include an attribute to indicate this condition.

A request with the "with-defaults" query parameter without a specified value will be interpreted as "report-all".

If we make a normal GET of the "sys/ntp/server" resource. The default values are not retrieved:

Example 21.15. The "sys/ntp/server" list (no defaults)

We can get also the default values by setting the query parameter `with-defaults=report-all`.

Example 21.16. The "sys/ntp/server" list with all defaults

21.3.2. Examples using POST, PUT and PATCH

To create a child resource you POST the child resource to its parent (target) resource, and in a successful result you will get returned the URI to the created resource in the `Location` header.

With PUT you will create or replace a target (data) resource. The message body is expected to contain the content used to create or replace the target resource.

Create a List Instance with POST

POST can be used to create a child resource to the resource specified by the uri. In this example we create a "route" list entry. The URI points to the parent container of the list. The payload in the POST request contain a "route" node that, at least, contain the list keys.

> **Note**
> When creating a list entry all keys must be given in the payload.

Example 21.17. Creating a "sys/routes/inet/route" resource

```
<route>
  <name>10.20.3.0</name>
  <prefix-length>24</prefix-length>
</route>
```

The server might respond:

When an element is successfully created the HTTP status response is 201. If the element already existed the status response is 409.

Create a Presence Container, within a list, with POST

In this example we create a "multilink" presence container. The URI points to the list instance that contains the presence container, i.e the URI ends with the key(s). The payload in the POST request contain a "multilink" node that contain a value for the leaf node "group".

Note

It is not possible to create a non-presence container with POST, as non-presence containers per definition always exists.

Example 21.18. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
<multilink>
  <group>1</group>
</multilink>
```

The server might respond:

Create and Replace a List Instance with PUT

PUT can be used to create or replace a resource. No distinction is made in the prerequisites for PUT. If no resource existed it is created, but if it existed it is replaced. In the example, note how the URI ends with the keys.

Example 21.19. Creating a "route" resource using PUT

```
<route>
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.4.4</name>
    <metric>100</metric>
  </next-hop>
</route>
```

The server might respond:

When an element is successfully created the HTTP status response is 201. We make a GET to verify the "route" list after the PUT of the element.

Example 21.20. The "route" resource after creation

> **Note**
> We didn't get the "metric" in the above result from the GET! This has to do with the (non) use of "deep" and "shallow", see: Section 21.2.10, “Query Parameters” [407]

We will now do a replace of the same "route" element.

Example 21.21. Replacing a "route" resource using PUT

```
<route>
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.3.1</name>
    <metric>100</metric>
  </next-hop>
  <next-hop>
    <name>192.168.4.2</name>
    <metric>200</metric>
  </next-hop>
</route>
```

The server might respond:

We get the 204 response as the resource already existed. We can verify that the element is replaced.

Example 21.22. The "route" resource after replace

Create and Replace Presence Container with PUT

In this example we create a "multilink" presence container. The uri points to the not yet existing URI for the presence container "multilink". The payload in the PUT request contain a "multilink" node that contain a value for the leaf node 'group'.

Example 21.23. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
<multilink>
  <group>1</group>
</multilink>
```

The server might respond:

Any subsequent request to the same URI will replace the content of "multilink" with the new payload. But the response code will instead be 204, since the resource already exists.

> **Note**
> Have in mind that PUT will replace everything with the provided payload. So in the example above, if the container "multilink" contained additional subnodes, other than "group", they would have been deleted as they were not present in the payload.

Replace Non-Presence Container with PUT

In this example we "populate" a non-presence container with subnode data. The URI points to the existing URI for the non-presence container "authentication", since a non-presence container always exist if its parent exist.

Example 21.24. Creating a "sys/interfaces/serial/ppp0/authentication" resource

```
<authentication>
  <method>pap</method>
  <list-name>foobar</list-name>
</authentication>
```

The server might respond:

> **Note**
> Have in mind that PUT will replace everything with the provided payload. So in the example above, if the container "authentication" contained additional subnodes, they would have been deleted as they were not present in the payload.

Example 21.25. The "authentication" resource after replace

Update Existing List Instance with PATCH

To update an existing resource the PATCH method can be used. PATCH will not be allowed on non-existent resources. When we use PATCH the payload should only contain the data that should be modified.

Example 21.26. Updating a "route" resource using PATCH

```
<route>
  <next-hop>
    <name>192.168.5.1</name>
    <metric>400</metric>
  </next-hop>
</route>
```

The server might respond:

The response status 204 indicated successful update. Here we can see that a new next-hop entry has been added to the route.

Example 21.27. The "route" resource after update

Update Presence Container with PATCH

PATCH can not be used to create new resources directly, since it must operate on existing resources. PATCH, in contrast to PUT, only merges the provided payload with the existing configuration, which makes it possible to create child resources within the target resource. In this example we create a "multilink" presence container. The uri points to the existing parent resource for the presence container "multilink".

Example 21.28. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
<serial>
  <multilink>
    <group>1</group>
  </multilink>
</serial>
```

The server might respond:

Update Non-Presence Container with PATCH

In this example we "populate" a non-presence container with subnode data. The URI points to the existing URI for the non-presence container "authentication", since a non-presence container always exist if its parent exist.

Example 21.29. Creating a "sys/interfaces/serial/ppp0/authentication" resource

```
<authentication>
  <method>eap</method>
</authentication>
```

The server might respond:

> **Note**
> Have in mind that PATCH will merge the existing configuration with the provided payload. So in the example above, if the container "authentication" contained additional subnodes not present in the payload, they will remain in the resulting configuration. Below the node 'list-name' was not present in the payload, but is still present in the resulting configuration.

Example 21.30. The "authentication" resource after update

Insert Data into Resources

For an ordered-by user list, the POST request can include the query parameters *insert* and *resource*.

The *insert* parameter indicated where the new element should be created. Legal values are:

- *first*: insert on top of list.
- *last*: insert on bottom of list.
- *before*: insert before the element indicated by the *resource* parameter.
- *after*: insert after the element indicated by the *resource* parameter.

The *resource* parameter contains the uri to an existing element in the list.

So we verify this functionality by first retrieving the "sys/dns/server" list:

Example 21.31. The "sys/dns/server" list before insert

We now insert a new element before the existing element:

Example 21.32. Insert=before in the "sys/dns/server" list

```
<server>
  <address>10.1.1.2</address>
</server>
```

The server might respond:

We get a 201 status when successful and we verify the result by retreiving the list once again:

Example 21.33. The "sys/dns/server" list after insert

21.3.3. Invoke Operations

To invoke an operation, use the POST method. The message body (if any) is processed as the operation input parameters.

Example 21.34. An "archive-log" action request example

The following yang model snippet shows the definition of the action *archive-log*:

```
     grouping syslog {
       list server {
         key "name";
         leaf name {
           type inet:host;
         }
         leaf enabled {
           type boolean;
         }
         list selector {
           key "name";
           leaf name {
             type int32;
           }
           leaf negate {
             type boolean;
           }
           leaf comparison {
             type enumeration {
               enum "same-or-higher";
               enum "same";
             }
           }
           leaf level {
             type syslogLevel;
           }
           leaf-list facility {
             type syslogFacility;
```

```
            min-elements 1;
            max-elements "8";
          }
        }
        leaf administrator {
          type string;
          tailf:hidden maint;
        }
        tailf:action archive-log {
          tailf:exec "./scripts/archive-log";
          input {
            leaf archive-path {
              type string;
            }
            leaf compress {
              type boolean;
            }
          }
          output {
            leaf result {
              type string;
            }
          }
        }
      }
    }
```

The action is invoked using the following URI. Note, the *_operations* tag that indicates the action invocation:

```

```

```
<input>
  <archive-path>/tmp</archive-path>
  <compress>false</compress>
</input>
```

The server might respond:

```

```

If the POST method succeeds, a "200 OK" Status-Line is returned if there is a response message body, and a "204 No Content" Status-Line is returned if there is no response message body.

If the user is not authorized to invoke the target operation, an error response containing a "403 Forbidden" Status-Line is returned to the client.

21.3.4. Delete Data Resources

A delete removes all data in the subtree under a resource. In this example we remove the complete "sys/interfaces/ex:serial" list.

Example 21.35. delete the "sys/interfaces/ex:serial" list

```

```

The server might respond:

The response status 204 indicates success. If we retrieve the "sys/interfaces" resource we can verify that the "ex:serial" list is removed.

Example 21.36. The "sys/interfaces" resource after delete

21.4. Resources

The RESTCONF protocol operates on a hierarchy of resources, starting with the top-level API resource itself. Each resource represents a manageable component within the device.

A resource can be considered a collection of conceptual data and the set of allowed methods on that data. It can contain child nodes that are nested resources. The child resource types and methods allowed on them are data-model specific.

A resource has its own media type identifier, represented by the `Content-Type` header in the HTTP response message. A resource can contain zero or more nested resources. A resource can be created and deleted independently of its parent resource, as long as the parent resource exists.

The RESTCONF resources are accessed via a set of URIs defined in this document. The set of YANG modules supported by the server will determine the additional data model specific operations, top-level data node resources, and notification event messages supported by the server.

The resources used in the RESTCONF protocol are identified by the "path" component in the request URI, see Example 21.2, “Request URI structure”. Each operation is performed on a target resource.

21.4.1. Representation

The RESTCONF protocol defines some application specific media types to identify each of the available resource types. The following resource types are defined in the REST API:

Table 21.3. Resources and their Media Types

Resource	Media Type
API	application/vnd.yang.api
Datastore	application/vnd.yang.datastore
Data	application/vnd.yang.data
Operation	application/vnd.yang.operation

Note

The REST API does not support all of the datastores defined in RESTCONF, neither does it use the same media types.
THIS DIFFERS FROM RESTCONF!

XML representation

A resource is represented in XML as an XML element, with an XML attribute "y:self" that contains the URI for the resource. In the XML representation, every resource has an XML attribute:

```
y:self="..."
```

Leafs are properties of the resource. They are encoded in XML as elements.

XML namespaces must be used whenever there are multiple sibling nodes with the same local name. This only happens if a YANG module augments a node with the same name as another node in the same container or list. XML namespaces MAY always be used, even if there are no risk of a conflict.

JSON representation

In the JSON representation, this URI is encoded as:

```
"_self": "..."
```

In the representation of a list resource, the keys are always present, and encoded first.

JSON doesn't have anything similar to XML namespaces. However, we have adopted the notion defined in the Internet Draft: *"Modeling JSON Text with YANG"*, where a *<yang-module-name>:<tag>* tag name is used in the JSON data to indicate the namespace. Note that it is only when the namespace changes that this notion is used.

Example 21.37. Namespaces in JSON

```
...
 "foo": [
   {
     "_self": "/api/operational/x/foo/1",
     "id": 1,
     // Note: the 'card' container exist in a different namespace
     //       compared to its parent 'foo' list element. The
     //       'a' is the Yang module name where 'card' is defined
     "a:card": {
       "_self": "/api/operational/x/foo/1/a:card",
       "id": 1
     }
...
```

21.4.2. API Resource (/api)

The top-level resource has the media type "application/vnd.yang.api+xml" or "application/vnd.yang.api +json". It is accessible through the well-known URI "/api".

This resource has the following fields:

Table 21.4. Fields of the /api resource

Field	Description
version	The version of the REST api.

Field	Description
config	Link to the "config" resource.
running	Link to the "running" resource.
startup	Link to the "startup" resource.
candidate	Link to the "candidate" resource.
operational	Link to the "operational" resource.
operations	Container for available operations (i.e: YANG rpc statements).
rollbacks	Container for available rollback files.

In XML, this resource is represented as an XML document with the document root element "y:api", underneath which all the resource's fields are represented as subelements.

In JSON, this resource is represented as a JSON object.

Supported HTTP methods: GET, HEAD, OPTIONS.

The version Field

The "version" field is a string identifying the version of the REST api.

The config Resource

The "config" resource represents a unified configuration datastore, used to simplify resource management for the client. The underlying NETCONF datastores are used to implement the unified datastore, but the clients do not have to care about which underlying datastore to used.

The server hides all NETCONF datastore details for edit operations, such as the :candidate and :startup capabilities. When a client writes to this resource, the server performs the edits in the datastores used; if the candidate is enabled, the changes are written to the candidate, and then the candidate is committed; if the startup is enabled, the changes are written to running and running is copied to startup.

The media type of this resource is either "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

Note

This resource resembles the RESTCONF "Datastore Resource" except that it does not contain operational data. RESTCONF differs from the REST API as it does not have separate NETCONF datastores for "running", "candidate" and "startup".

The running Resource

The "running" resource represents the running configuration datastore, and is present on all devices. Not all devices support direct modification to this resource.

The media type of this resource is either "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

The startup Resource

The "startup" resource represents the startup configuration datastore. Not all devices support this resource.

The media type of this resource is "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

The candidate Resource

The "candidate" resource represents the candidate configuration datastore. Not all devices support this resource.

The media type of this resource is "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

The operational Resource

The "operational" read-only resource represents the state data as well as the config data on the device, and is present on all devices. Note that actions defined as *config false* also will show up in this resource.

The media type of this resource is "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

Note

This resource resembles the RESTCONF "Datastore Resource" except that it is read-only.

The operations Container

The "operations" container contains all operations defined with the "rpc" statement in the YANG models supported on the device.

The rollbacks Container

The "rollbacks" container contains all available rollback files to be used to rollback the configuration state to an earlier incarnation.

Examples

In order to retrieve the representation of this resource, a client can send:

Note the use of the 'verbose' query parameter, see Section 21.2.10, “Query Parameters”.

Example 21.38. GET the /api resource

```
```

The server might respond:

```
```

21.4.3. Datastore Resource (/api/<datastore>)

The media types "application/vnd.yang.datastore+xml" and "application/vnd.yang.datastore+json" represent a complete datastore. All three configuration datastores defined by NETCONF are available, as the resources:

- running
- candidate

- startup
- operational (read-only)

> **Note**
> The state data defined by NETCONF is available as a read-only resource "operational".
> *THIS DIFFERS FROM RESTCONF!*

A *config* datastore (i.e one of "running", "candidate","startup") resource has the following fields:

Table 21.5. Fields of the /api/<datastore> resource

Field	Description
operations	Container for available built-in operations.
y:operations	Container for available user defined actions.
<all top-level data models nodes>	Top-level nodes from the YANG models.

The *"operational"* datastore contains both *operational* and *config* data; as well as the containers as shown in the table: Table 21.5, “Fields of the /api/<datastore> resource”.

The REST API has a number of built-in operations that may be applicable for a particular datastore, and if they are applicable they are included in the *operations* container as described below:

Table 21.6. Built in operations

Field	Description
commit	Link to the "commit" resource.
copy-running-to-startup	Link to the "copy-running-to-startup" resource.
discard-changes	Link to the "discard-changes" resource.
lock	Link to the "lock" resource.
validate	Link to the "validate" resource.

In XML, the */api/<datastore>* resource is represented as an XML document with the document root element "y:data", underneath which all the resource's fields are represented as subelements.

In JSON, this resource is represented as a JSON object.

The operations are described below, and all represented by the media type "application/vnd.yang.operation", see Section 21.3.3, “Invoke Operations”.

The lock Resource

In order to access a datastore through a lock, the client needs to POST to the "lock" resource. If the server is able to lock the datastore, the POST request succeeds with the status code "201 Created", and the "Location" HTTP header contains the URI to a newly created resource representing the locked datastore. To unlock the datastore, the client deletes the newly created resource using the HTTP method DELETE.

The "config" resource cannot be locked.

In the representation of a locked datastore, the "lock" operation is not available.

In order to facilitate recovery from failing clients with outstanding locks, the REST server deletes the resource representing the lock after some time of inactivity.

The media type of this resource is "application/vnd.yang.operation+xml".

The commit Resource

In the representation of the candidate datastore, the "commit" operation is present, and can be POSTed to by the client to commit candidate to running, as described in section 8.3 in RFC 6241.

The media type of this resource is "application/vnd.yang.operation".

The copy-running-to-startup Resource

In the representation of the running datastore, the "copy-running-to-startup" operation is present, if the server also supports the startup datastore, and can be POSTed to by the client to copy the contents of running to startup, as described in section 8.7 in RFC 6241.

The media type of this resource is "application/vnd.yang.operation".

The discard-changes Resource

In the representation of the candidate datastore, the "discard-changes" operation is present, and can be POSTed to by the client to revert the candidate to the current running configuration, as described in section 8.3.4.2 in RFC 6241.

The media type of this resource is "application/vnd.yang.operation".

The validate Resource

In the representation of the candidate datastore, the "validate" operation is present, and can be POSTed to by the client to validate the contents of the candidate, as described in section 8.6.4.1 in RFC 6241.

The media type of this resource is "application/vnd.yang.operation".

Examples

The examples in this section could instead have been performed using JSON specific mime types such as "application/vnd.yang.api+json", "application/vnd.yang.datastore+json" and "application/vnd.yang.data+json".

To retrieve a representation of the running datastore in XML format, a client can send:

(Note the use of the 'verbose' query parameter, see Section 21.2.10, “Query Parameters”)

Example 21.39. GET the /api/running resource

The server might respond:

To copy running to startup, a client can send:

```
POST /api/running/_copy-running-to-startup
Host: example.com
Accept: application/vnd.yang.operations+xml
```

Note that any action defined as "config false" will only show up under the */api/operational* datastore.

Example 21.40. Action in /api/operational

```
...
<blaha xmlns="http://example.com/ns/ktm"
       y:self="/api/operational/blaha">
  <y:operations y:path="/blaha">
    <bar y:self="/api/operational/blaha/_operations/bar"/>
    <!-- NOTE: this action is defined as 'config false'
         and hence, shows up under the 'operational'
         datastore  -->
  </y:operations>
</blaha>
...
```

21.4.4. Data Resource (/api/<datastore>/<data>)

By default, all top-level objects, list entries, and containers are resources. Any resource derived from a YANG module is represented with the media type "application/vnd.yang.data+xml".

When such a resource is retrieved, a "path" property is included in its representation (a "y:path" attribute in XML). The value of this property is the resource's instance-identifier.

Supported HTTP methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT.

Note that resources representing non-presence containers cannot be deleted, and thus they do not support the DELETE method.

Refer to Section 21.3, “Resource Examples” for examples of how to operate on "application/vnd.yang.data +xml".

21.4.5. Operations and Actions

YANG-defined operations, defined with the YANG statements "rpc" or "tailf:action", and the built-in operations, are represented with the media type "application/vnd.yang.operation".

Resources of this type accept only the method "POST".

In XML, such resources are encoded as subelements to the XML element "y:operations". In JSON, they are encoded under "_operations".

If an operation does not require any parameters, the POST message has no body. If the client wishes to send parameters to the operation, they are encoded as an XML document with the document element "input".

If an operation does not produce any output, the HTTP response code is 204 (No Content). If it produces output, the HTTP response code is 200 (OK), and the output of the operation is encoded as an XML document with the document element "output".

Supported HTTP methods: POST

21.4.6. The Rollback Resource

The rollback resource can be accessed from the top level "/api/rollbacks" resource as described above, and a rollback file can be applied to any database.

Listing and Inspecting Rollback Files

In order to list available a rollback files, a client can send:

Example 21.41. GET rollback files information

The server might respond:

Note how each rollback file is represented as separate resources, e.g. "/api/rollbacks/0". These resources can be inspected individually and a client can send:

Example 21.42. GET rollback file content

The server might respond:

The payload is in the same curly bracket rollback format as used in the NETCONF, CLI and Web UI agents.

Applying Rollback Files

To apply a rollback file to a database use the appropriate "rollback" resource/operation in the datastore of your choice:

Example 21.43. Find and use the rollback operation resource

The server might respond:

Note the "/api/running/_rollback" resource operation.

POST an appropriate rollback file name to the "/api/running/_rollback" resource operation to apply it:

The server might respond:

21.5. Configuration Meta-Data

As described in Chapter 8, *Configuration Meta-Data* it is possible to associate meta-data with the configuration data. For REST, resources such as containers, lists as well as leafs and leaf-lists can have such meta-data. For XML, this meta-data is represented as attributes, attached to the XML element in question. For JSON, there does not exist a natural way to represent this info. Hence we have introduced a special notation, see the example below.

Example 21.44. XML representation of meta-data

```
<x xmlns="urn:x"
   y:self="/api/running/x"
   xmlns:y="http://tail-f.com/ns/rest"
   xmlns:x="urn:x"
   y:path="/x:x">
  <id tags=" important ethernet " annotation="hello world">42</id>
  <person y:self="/api/running/x/person" annotation="This is a person">
    <name>Bill</name>
    <person annotation="This is another person">grandma</person>
  </person>
</x>
```

Example 21.45. JSON representation of meta-data

```
{
  "x": {
    "_self": "/api/running/x",
    "_path": "/x:x",
    "id": 42,
    "@id": {"tags": ["important","ethernet"],"annotation": "hello world"},
    "person": {
      "_self": "/api/running/x/person",
      // NB: the below refers to the parent object
      "@@person": {"annotation": "This is a person"},
      "name": "Bill",
      "person": "grandma",
      // NB: the below refers to the sibling object
      "@person": {"annotation": "This is another person"}
    }
  }
}
```

For JSON, note how we represent the meta data for a certain object "x" by another object constructed of the object name prefixed with either one or two "@" signs. The meta-data object "@x" refers to the sibling object "x" and the "@@x" object refers to the parent object.

Note

THIS DIFFERS FROM RESTCONF!

21.6. Request/Response headers

There are some optional request and response headers that are of interest since some functionality is obtained by their use. We here focus on the `Etag` and `Last-Modified` response headers, and the request

headers that are correlated to these (`If-Match`, `If-None-Match`, `If-Modified-Since` and `If-Unmodified-since`).

21.6.1. Response headers

- `Etag`: This header (entity-tag) is a string representing the latest transaction id in the database. This header is only available for the "running" resource or equivalent.
- `Last-Modified`: This header contains the timestamp for the last change in the database. This header is only available for the "running" resource or equivalent. Also, this header is only available if rollback files are enabled.

21.6.2. Request headers

- `If-None-Match`: This header evaluates to true if the supplied value does not match the latest `Etag` value. If evaluated to false an error response with status 304 (Not Modified) will be sent with no body. The usage of this is for instance for a GET operation to get information if the data has changed since last retrieval. This header carry only meaning if the `Etag` response header has previously been acquired.
- `If-Modified-Since`: This request-header field is used with a HTTP method to make it conditional, i.e if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (Not Modified) response will be returned without any message-body.

  Usage of this is for instance for a GET operation to get information if (and only if) the data has changed since last retrieval. Thus, this header should use the value of a `Last-Modified` response header that has previously been acquired.
- `If-Match`: This header evaluates to true if the supplied value matches the latest `Etag` value. If evaluated to false an error response with status 412 (Precondition Failed) will be sent with no body.

  The usage of this can be to control if a POST operation should be executed or not (i.e. do not execute if the database has changed). This header carry only meaning if the `Etag` response header has previously been acquired.
- `If-Unmodified-Since`: This header evaluates to true if the supplied value is later or equal to the last acquired `Last-Modified` timestamp. If evaluated to false an error response with status 412 (Precondition Failed) will be sent with no body.

  The usage of this can be to control if a POST operation should be executed or not (i.e. do not execute if the database has changed). This header carry only meaning if the `Last-Modified` response header has previously been acquired.

21.7. Special characters

When setting or retrieving data it is sometimes necessary to represent special characters in the payload. In the REST api the payload can have both XML and JSON format. The special characters handled in the REST api are:

- "new line": representing a line feed (decimal ascii value 10)
- "carriage return": representing carriage return (decimal ascii value 13)
- "horizontal tab": representing a tabulation (decimal ascii value 9)

The ambition in the REST api is that special characters should be handled in the same way as they are in the CLI. Since the CLI is capable to present configuration data both as strings and XML the CLI representation can be used as template for both the XML and JSON format.

21.7.1. XML representation of special characters

When in the XML case the special characters uses the representation &#xH; where H is the ascii hex value or &#DD; where DD is the ascii decimal value. Since "&" is the quoting character this is also treated as a special character, and so is "<" since this is the separator character indicating the beginning of a tag value. The following is the list of XML special characters:

- "new line": represented by
 or

- "carriage return": represented by
 or
- "horizontal tab": represented by 	 or 	
- "&": represented by &
- "<": represented by <

An example XML fragment is

```
<foo>123\n456\\n &lt;&amp;></foo>

 which is interpreted as:

123
456\n <&>
```

21.7.2. JSON representation of special characters

In the JSON formatted string case the quote character "\" and string separator characters are used "\"" which also becomes special characters in the string case. The complete list of JSON special characters become:

- "new line": represented by \n
- "carriage return": represented by \r
- "horizontal tab": represented by \t
- "\": represented by \ or \\
- """: represented by \"

The \ quote character is used in the following way: A single \ in a string that is not directly followed by characters n,r or t are unaltered (as a \ character). Two \\ are interpreted as \ (the escaped \). This implies that both \\\ and \\\\ are are interpreted as \\ and so forth.

An example JSON string is

```
```

```
"123\n456\\n \\\"

 which is interpreted as:

123
456\n \\
```

21.8. Error Responses

Error responses are formatted either in XML and JSON depending on the preferred MIME type in the request.

In your installed release you should be able to find a Yang file named `tailf-rest-error.yang` that defines the structure of these error replies. An easy way to find the file run, from the top directory of your installation:

```
     find . -name tailf-rest-error.yang
```

Let's start by looking at an example of how a structured error reply can look like.

Example 21.46. Example of a XML formatted error message

```
 curl -i -X PUT "localhost:8008/api/running/hosts"
-H "Content-Type: application/vnd.yang.data+xml" ...
HTTP/1.1 500 Internal Server Error
Server: ConfD/5.3.0
Cache-control: private, no-cache, must-revalidate, proxy-revalidate
Date: Thu, 10 Jul 2014 07:59:04 GMT
Content-Length: 259
Content-Type: text/xml


<errors xmlns="http://tail-f.com/ns/tailf-rest-error">
  <error>
    <error-tag>operation-failed</error-tag>
    <error-urlpath>/api/running/hosts</error-urlpath>
    <error-message>internal error</error-message>
  </error>
</errors>
```

Example 21.47. Example of a JSON formatted error message

```
 curl -i -X POST "localhost:8008/api/running/hosts2" \
-H "Content-Type: application/vnd.yang.data+json" ...
HTTP/1.1 400 Bad Request
Server: ConfD/5.3.0
Cache-control: private, no-cache, must-revalidate, proxy-revalidate
Date: Thu, 10 Jul 2014 09:11:13 GMT
Content-Length: 199
Content-Type: text/json
```

```
{"errors":
  {"error":
     [{
       "error-message": "unexpected trailing data: hosts",
       "error-urlpath": "/api/running/hosts",
       "error-tag": "malformed-message"
       }
      ]
  }
}
```

The YANG model for the error messages is taken from the NETCONF specification. However, note that the REST API currently only sets three values when reporting an error:

- "error-tag": An error classification tag.
- "error-urlpath": The URI used by the error generating request.
- "error-message": A descriptive error message.

The error-tag element has a number of predefined values and there is also a preferred HTTP status code connected to each error-tag. These are:

- "in-use": HTTP Status: `409 Conflict`
- "invalid-value": HTTP Status: `400 Bad Request`
- "too-big": HTTP Status: `413 Request Entity Too Large`
- "missing-attribute": HTTP Status: `400 Bad Request`
- "bad-attribute": HTTP Status: `400 Bad Request`
- "unknown-attribute": HTTP Status: `400 Bad Request`
- "bad-element": HTTP Status: `400 Bad Request`
- "unknown-element": HTTP Status: `400 Bad Request`
- "unknown-namespace": HTTP Status: `400 Bad Request`
- "access-denied": HTTP Status: `403 Forbidden`
- "lock-denied": HTTP Status: `409 Conflict`
- "resource-denied": HTTP Status: `409 Conflict`
- "rollback-failed": HTTP Status: `500 Internal Server Error`
- "data-exists": HTTP Status: `409 Conflict`
- "data-missing": HTTP Status: `409 Conflict`
- "operation-not-supported": HTTP Status: `501 Not Implemented`
- "operation-failed": HTTP Status: `500 Internal Server Error`
- "partial-operation": HTTP Status: `500 Internal Server Error`

- "malformed-message": HTTP Status: `400 Bad Request`

21.8.1. User extended error messages

For data providers, hooks, transforms etc there exist a possibility to add extensions to error messages and change error codes before sending errors back to the server from the callbacks (see: Section 27.14, “Error Message Customization”). These codes and error messages will also be visible over the REST interface. More on how to use these options can be found in the confd_lib_dp(3) man page, e.g under the `confd_db_seterr_extended` function, or in the Javadoc for the `com.tailf.dp.DpCallbackExtendedException` class.

Using the above mechanism to change the errorcode for an emitted error, will have effect on the REST HTTP response statuses. The following table show their relationship:

Table 21.7. Error code vs HTTP Status

Error Code	HTTP Status
CONFD_ERRCODE_IN_USE	409 Conflict
CONFD_ERRCODE_RESOURCE_DENIED	409 Conflict
CONFD_ERRCODE_INCONSISTENT_VALUE	400 Bad Request
CONFD_ERRCODE_ACCESS_DENIED	403 Forbidden
CONFD_ERRCODE_APPLICATION	400 Bad Request
CONFD_ERRCODE_APPLICATION_INTERNAL	500 Internal Server Error
CONFD_ERRCODE_DATA_MISSING	409 Conflict
CONFD_ERRCODE_INTERRUPT	500 Internal Server Error

21.9. The Query API

The Query API consists of a number of Requests and Replies which are sent as payload via the (REST) HTTP connection.

In your installed release you should be able to find two YANG files named `tailf-rest-query.yang` and `tailf-common-query.yang` that defines the structure of these Requests / Replies. An easy way to find the files is to run the following command from the top directory of your installation:

```
find . -name tailf-rest-query.yang
```

The API consists of the following Requests:

- "start-query": Start a query and return a query handle.
- "fetch-query-result": Use a query handle to repeatedly fetch chunks of the result.
- "reset-query": (Re)set where the next fetched result will begin from.
- "stop-query": Stop (and close) the query.

The API consists of the following Replies:

- "start-query-result": Reply to the start-query request

- "query-result": Reply to the fetch-query-result request

In the following examples, we'll use this data model:

```
container x {
  list host {
    key number;
    leaf number {
      type int32;
    }
    leaf enabled {
      type boolean;
    }
    leaf name {
      type string;
    }
    leaf address {
      type inet:ip-address;
    }
  }
}
```

The actual format of the payload should be represented either in XML or JSON. For XML it could look like this:

```
   <start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
     <foreach>
       /x/host[enabled = 'true']
     </foreach>
     <select>
       <label>Host name</label>
       <expression>name</expression>
       <result-type>string</result-type>
     </select>
     <select>
       <expression>address</expression>
       <result-type>string</result-type>
     </select>
     <sort-by>name</sort-by>
     <limit>100</limit>
     <offset>1</offset>
   </start-query>
```

An informal interpretation of this query is:

For each '/x/host' where 'enabled' is true, select its 'name', and 'address', and return the result sorted by 'name', in chunks of 100 results at the time.

Let us discuss the various pieces of this request. To start with, when using XML, we need to specify the name space as shown:

```
   <start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
```

The actual XPath query to run is specified by the 'foreach' element. In the example below will search for all '/x/host' nodes that has the 'enabled' node set to 'true':

```
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
```

Now we need to define what we want to have returned from the node set by using one or more 'select' sections. What to actually return is defined by the XPath 'expression'.

Choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk The possible result-types are: 'string' , 'path' , 'leaf-value' and 'inline'.

The difference between 'string' and 'leaf-value' is somewhat subtle. In the case of 'string' the result will be processed by the XPath function: string() (which if the result is a node-set will concatenate all the values). The 'leaf-value' will return the value of the first node in the result. As long as the result is a leaf node, 'string' and 'leaf-value' will return the same result. In the example above, the 'string' is used as shown below. At least one result-type must be specified.

The result-type 'inline' makes it possible to return the full sub-tree of data, either in XML or in JSON format. The data will be enclosed with a tag: 'data'.

It is possible to specify an optional 'label' for a convenient way of labeling the returned data:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as XPath expressions, which in most cases are very simple and refers to the found node set. In this example we sort the result by the content of the 'name' node:

```
  <sort-by>name</sort-by>
```

To limit the max amount of results in each chunk that 'fetch-query-result' will return we can set the 'limit' element. The default is to get all results in one chunk.

```
<limit>100</limit>
```

With the 'offset' element we can specify at which node we should start to receive the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

This request, expressed in JSON, would look like this:

```
{
 "start-query": {
   "foreach": "/x/host[enabled = 'true']",
   "select": [
     {
       "label": "Host name",
       "expression": "name",
       "result-type": ["string"]
     },
     {
       "expression": "address",
       "result-type": ["string"]
     }
   ],
   "sort-by": ["name"],
   "limit": 100,
   "offset": 1
 }
}
```

Now, if we continue by putting this XML example in a file `test.xml` we can send a request, using the command 'curl', like this:

```
curl -i 'http://admin:admin@localhost:8008/api/query' \
     -X POST -T test.xml \
     -H "Content-Type: application/vnd.yang.data+xml"
```

The important parts of the above is the '/api/query' in the URI and that we send a HTTP 'POST' with the correct 'Content-Type'.

The result would look something like this:

```
<start-query-result>
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example '12345') must be used in all subsequent calls. To retrieve the result, we can now send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
```

```
        <value>10.0.0.1</value>
      </select>
    </result>
    <result>
      <select>
        <label>Host name</label>
        <value>Three</value>
      </select>
      <select>
        <value>10.0.0.3</value>
      </select>
    </result>
  </query-result>
```

If we try to get more data with the 'fetch-query-result' we might get more 'result' entries in return until no more data exists and we get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
</query-result>
```

If we want to go back in the "stream" of received data chunks and have them repeated, we can do that with the 'reset-query' request. In the example below we ask to get results from the 42:nd result entry:

```
<reset-query xmlns=\"http://tail-f.com/ns/tailf-rest-query\">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

Finally, when we are done we stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</stop-query>
```

21.10. Custom Response HTTP Headers

The REST server can be configured to reply with particular HTTP headers in the HTTP response. For example, to support Cross-Origin Resource Sharing (CORS, http://www.w3.org/TR/cors/) there is a need to add a couple of headers to the HTTP Response.

We add the extra configuration parameter in `confd.conf`

Example 21.48. ConfD configuration for REST

```
<rest>
  <enabled>true</enabled>
  <customHeaders>
    <header>
      <name>Access-Control-Allow-Origin</name>
      <value>*</value>
```

```
        </header>
      </customHeaders>
    </rest>
```

Example 21.49.

Send a request with Origin header:

```
OPTIONS .bob.com
```

A result can then look like

```
HTTP/1.1 200 OK
Server: ConfD/5.4.0
Allow: GET, HEAD
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 0
Content-Type: text/html
Access-Control-Allow-Origin: http://api.bob.com
Pragma: no-cache
```

21.11. HTTP Status Codes

The REST server will return standard HTTP response codes, as described in the list below:

200 OK	The request was successfully completed, and a response body is returned containing a representation of the resource.
201 Created	A resource was created, and the new resource URI is returned in the "Location" header.
204 No Content	The request was successfully completed, but no response body is returned.
400 Bad Request	The request could not be processed because it contains missing or invalid information (such as validation error on an input field, a missing required value, and so on).
401 Unauthorized	The request requires user authentication. The response includes a "WWW-Authenticate" header field for basic authentication.
403 Forbidden	Access to the resource was denied by the server, due to authorization rules.
404 Not Found	The requested resource does not exist.
405 Method Not Allowed	The HTTP method specified in the request (DELETE, GET, HEAD, PATCH, POST, PUT) is not supported for this resource.
406 Not Acceptable	The resource identified by this request is not capable of generating the requested representation, specified in the "Accept" header or in the "format" query parameter.

409 Conflict	This code is used if a request tries to create a resource that already exists.
415 Unsupported Media Type	The format of the request is not supported.
500 Internal Error	The server encountered an unexpected condition which prevented it from fulfilling the request.
501 Not Implemented	The server does not (currently) support the functionality required to fulfill the request.
503 Unavailable	The server is currently unable to handle the request due to the resource being used by someone else, or the server is temporarily overloaded.

Chapter 22. The Management Agent API

22.1. What is MAAPI?

MAAPI is a C API which provides full access to the ConfD internal transaction engine. It is used in a number of different settings.

- We use MAAPI if we want to write our own management application. Using the MAAPI interface, it is for example possible to implement a custom built command line interface (CLI) or Web UI. This usage is described below.
- We use MAAPI to access ConfD data inside a not yet committed transaction when we wish to implement semantic validation in C. This is fully described in Chapter 9, *Semantic validation*.
- We use MAAPI to access ConfD data inside a not yet committed transaction when we wish to implement CLI wizards in C. Here we can invoke an external C program which can read and write, both to the executing transaction, but also interact with the CLI user. This is fully described in Chapter 16, *The CLI agent*.
- Finally MAAPI is also used during database upgrade to access and write data to a special upgrade transaction. This is fully described in Chapter 5, *CDB - The ConfD XML Database*.

Thus, MAAPI is an API which contains functions that are the northbound equivalents of all the callbacks described in Chapter 7, *The external database API*.

A typical sequence of API calls when using MAAPI to write a management application would be

1. Create a user session, this is the equivalent of an established SSH connection from a NETCONF manager. It is up to the MAAPI application to authenticate users. The TCP connection from the MAAPI application to ConfD is a clear text connection.
2. Establish a new ConfD transaction
3. Issue a series of read and write operations towards the ConfD transaction
4. Commit or abort the transaction

MAAPI also has support for several operations that do not work immediately towards a ConfD transaction. This includes users session management, locking, and candidate manipulation.

22.2. A custom toy CLI

In this section we implement a small toy CLI towards a specific data model. We start with the following YANG module:

Example 22.1. scli.yang YANG module

```
module scli {
  namespace "http://tail-f.com/test/scli";
  prefix scli;
```

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

  leaf foo {
    type string;
    default "fooo";
  }
  leaf bar {
    type int32;
    default 66;
  }
  list servers {
    key "name";
    max-elements 64;
    leaf name {
      type string;
    }
    leaf ip {
      type inet:ipv4-address;
      mandatory true;
    }
    leaf port {
      type inet:port-number;
      mandatory true;
    }
  }
}
```

We compile this file with confdc (1) and load it into ConfD as usual. Data is kept in CDB, since there are no callpoints defined. At this point, we can manipulate the configuration data with a NETCONF manager or the ConfD built-in CLI.

What we wish to do is to write a small `custom` CLI, that understands the structure of the XML tree using the MAAPI.

All functions in MAAPI are described in the confd_lib_maapi(3) manual page. We will use a subset of those functions in this example.

We start off with some global variables, and a trivial `main()` function.

```
#include "confd_lib.h"
#include "confd_maapi.h"

/* include .h file generated by confdc */
#include "scli.h"

static int sock, th;
static char *user = "admin";
static const char *groups[] = {"admin"};
static int debuglevel = CONFD_DEBUG;
static char *context = "maapi";
static enum confd_dbname dbname = CONFD_RUNNING;

int main(int argc, char **argv)
{
    int c;
    while ((c = getopt(argc, argv, "rc")) != -1) {
        switch(c) {
```

```
        case 'r':
            dbname = CONFD_RUNNING;
            break;
        case 'c':
            dbname = CONFD_CANDIDATE;
            break;

        }
    }
    confd_init("MYNAME", stderr, debuglevel);
    cnct();
    runcli();
}
```

The code in `main()` sets a global variable `dbname` which will be the database we are opening, i.e. we can choose to run this CLI either towards the running database or towards the candidate database.

The code first calls `cnct()` which creates a MAAPI socket and connects to ConfD, following that we run the simple CLI.

The code to connect to ConfD looks like this:

```
static void cnct()
{
    struct in_addr in;
    struct sockaddr_in addr;
    struct confd_ip ip;

    inet_aton("127.0.0.1", &in);
    addr.sin_addr.s_addr = in.s_addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4565);

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

    if (maapi_connect(sock, (struct sockaddr*)&addr,
                      sizeof (struct sockaddr_in)) < 0)
        confd_fatal("Failed to confd_connect() to confd \n");

    ip.af = AF_INET;
    inet_aton("66.55.44.33", &ip.ip.v4);

    maapi_start_user_session(sock, user, context, groups, 1,
                             &ip, CONFD_PROTO_TCP);

    th = maapi_start_trans(sock, dbname, CONFD_READ_WRITE);
    maapi_set_namespace(sock, th, scli__ns);
}
```

The code first creates a normal socket, and then connects to ConfD using `maapi_connect()`. Following that the code calls `maapi_start_user_session()` which creates a `user session` on the socket.

We must usually have an established user session before we can issue any of the MAAPI calls. There are some calls that can be performed on the socket without an already established user session.

A user session must be authenticated by the agent before the agent connects to ConfD. This authentication must be performed by the agent code, and once the agent issues the call to `maapi_start_user_session()`, the authentication must already be done. ConfD also has an au-

thentication system built-in to it. The behavior of the ConfD built-in authentication system is controlled through the AAA section in `confd.conf`. Thus, when we're implementing a proprietary agent, either we do the authentication ourselves, or we can ask ConfD to authenticate the user through the API function `maapi_authenticate()`. Thus we could have written:

```
char *groups[10];
if (maapi_authenticate(sock, user, pass, &groups[0], 10) == 1) {
    int i = 0;
    while (groups[i])
      i++;
    maapi_start_user_session(sock, user, context, groups, i,
                             &srcip, CONFD_PROTO_TCP);
```

This way we only need to obtain the clear text password in the agent and we can let ConfD perform the actual authentication as well as the group assignment. The group assignment is important, since the authorization model in ConfD hinges entirely on group membership.

Once we have established a user session, we continue to create a transaction towards either the running database or the candidate. This is done through the call to `maapi_start_trans()`. The call returns a transaction handle, a `th` which is subsequently used when we read and write data. Remember that we are creating a transaction, thus nothing is written to actual storage until we commit the transaction. This applies also when we access the running database.

Finally we make our first call towards the transaction, and that is to indicate the name of the namespace we are planning to work against. This is a hashed integer value which can be found in a header file generated from `scli.fxs`. Read more about how to generate a header (.h) file from a .fxs file in the confdc (1) compiler man page.

Once we have connected, established a user session and also created our first transaction, we enter the CLI input loop called `runcli()`.

```
#define DELIM " \n\r"

static void runcli()
{
    char ibuf[BUFSIZ];
    struct maapi_cursor mc;
    int ival;
    char *tok;

    printf("#cli "); fflush(stdout);

    while(fgets(ibuf, BUFSIZ, stdin) != NULL) {
        if ((tok = strtok(ibuf, DELIM)) == NULL) {
            printf("#cli "); fflush(stdout);
            continue;
        }

        if (strcmp(tok, "show") == 0) {
            if (maapi_get_str_elem(sock, th, ibuf, BUFSIZ, "/foo") ==CONFD_OK)
                printf ("foo: %s\n", ibuf);
            else if (confd_errno == CONFD_ERR_NOEXISTS)
                printf ("foo: \n");
            else
                confd_fatal("What \n");

            if (maapi_get_int32_elem(sock, th, &ival, "/bar") == CONFD_OK)
                printf ("bar: %d\n", ival);
```

```
            else if (confd_errno == CONFD_ERR_NOEXISTS)
                printf ("bar: ");
            else
                confd_fatal("What \n");

            maapi_init_cursor(sock, th, &mc, "/servers/server");
            maapi_get_next(&mc);
            while (mc.n != 0) {
                struct in_addr ip;
                uint16_t port;
                char tmpbuf[BUFSIZ];

                maapi_get_ipv4_elem(sock, th, &ip,
                                    "/servers/server{%x}/ip",
                                    &mc.keys[0]);
                maapi_get_u_int16_elem(sock, th, &port,
                                       "/servers/server{%x}/port",
                                       &mc.keys[0]);
                confd_pp_value(tmpbuf,BUFSIZ,&mc.keys[0]);
                printf ("server name=%s ip=%s port=%d\n",
                        tmpbuf, inet_ntoa(ip), port);
                maapi_get_next(&mc);
            }
            maapi_destroy_cursor(&mc);
        }
        else if (strcmp(tok, "abort") == 0) {
            maapi_abort_trans(sock, th);
            maapi_finish_trans(sock, th);
            th = maapi_start_trans(sock, dbname, CONFD_READ_WRITE);
            maapi_set_namespace(sock, th, scli__ns);
        }
        else if (strcmp(tok, "commit") == 0) {
            maapi_apply_trans(sock, th, 0);
            maapi_finish_trans(sock, th);
            th = maapi_start_trans(sock, dbname, CONFD_READ_WRITE);
            maapi_set_namespace(sock, th, scli__ns);
        }
        else if (strcmp(tok, "create") == 0) {
            char *name; char *ipstr; char *portstr;
            if (((name = strtok(NULL, DELIM)) == NULL) ||
                ((ipstr = strtok(NULL, DELIM)) == NULL) ||
                ((portstr = strtok(NULL, DELIM)) == NULL)) {
                printf ("input error \n"); goto err;
            }
            if (maapi_create(sock, th, "/servers/server{%s}", name) !=
                CONFD_OK) {
                printf ("error: %d %s \n ", confd_errno,confd_lasterr());
                goto err;
            }
            if (maapi_set_elem2(sock,th,ipstr,"/servers/server{%s}/ip",name)
                != CONFD_OK) {
                printf ("error: %d %s \n ", confd_errno, confd_lasterr());
                goto err;
            }
            if (maapi_set_elem2(sock,th,portstr,"/servers/server{%s}/port",
                                name)
                != CONFD_OK) {
                printf ("error: %d %s \n ", confd_errno, confd_lasterr());
                goto err;
            }
```

```
        }
        else if (strcmp(tok, "delete-config") == 0) {
            if (maapi_delete_config(sock, dbname) != CONFD_OK)
                printf ("error: %d, %s\n", confd_errno,
                        confd_lasterr());
        }

        else if (strcmp(tok, "delete") == 0) {
            char *name;
            if ((name= strtok(NULL, DELIM)) == NULL) {
                printf ("input error"); goto err;
            }
            if (maapi_delete(sock, th, "/servers/server{%s}", name)
                != CONFD_OK) {
                printf ("error: %s \n ", confd_lasterr());
                goto err;
            }
        }
        else if (strcmp(tok, "candidate-reset") == 0) {
            if (maapi_candidate_reset(sock) != CONFD_OK)
                printf ("error: %d, %s\n", confd_errno,
                        confd_lasterr());
        }

        else if (strcmp(tok, "validate-trans") == 0) {
            if (maapi_validate_trans(sock, th,1,1) == CONFD_OK) {
                printf ("ok \n");
            }
            else {
                printf ("nok: %s\n", confd_lasterr());
            }
        }
        else if (strcmp(tok, "candidate-confirmed-commit") == 0) {
            char *istr = strtok(NULL, DELIM);
            if (!istr) {printf("input error\n"); goto err;}
            if (maapi_candidate_confirmed_commit(sock, atoi(istr))!=CONFD_OK) {
                printf("error: %s\n", confd_lasterr());
            }
            else {
                printf("ok\n");
            }
        }
        else if (strcmp(tok, "candidate-commit") == 0) {
            if (maapi_candidate_commit(sock) != CONFD_OK) {
                printf("error: %s\n", confd_lasterr());
            }
            else {
                printf("ok\n");
            }
        }
        else {
            printf("commands \n");
            printf("   show    - show current conf\n");
            printf("   help    - show this text\n");
            printf("   abort   - abort current trans \n");
            printf("   commit  - commit current trans\n");
            printf("   create name ip port  - create new server\n");
            printf("   delete name          - delete server\n");
            printf("   candidate-reset      - copy running into cand");
            printf("   validate             - trans validate");
```

```
            printf("   delete-config        - delete config");
            printf("   candidate-commit     - copy cand to running | confirm");
            printf("   candidate-confirmed-commit secs \n");

            printf(" \n");
        }
          err:
        printf("#cli "); fflush(stdout);
    }
}
```

The code above reads lines from stdin, parses the line and invokes different MAAPI calls. For example, if we wish to use the **show** command to show the contents of the opened database, we first read the two leaf elements, called `/foo` and `/bar`. We must check the return values from those read calls. If we look at the data model, the values are defined as:

```
  leaf foo {
    type string;
    default "fooo";
  }
  leaf bar {
    type int32;
    default 66;
  }
```

The above elements both have default values. However if we open an empty candidate, they do not exist there.

Following that, in order to show the database, we must traverse all `/servers/server` instances and display them on the screen. We do this by using a `maapi_cursor`. A cursor must first be initialized, and then we can use the cursor to find out the key value(s) of the `next` element until there are no more elements.

Another interesting call is the call to create a new server. We create a new server instance through the call to `maapi_create(sock, th, "/servers/server{%s}", name)`. Once we have created a new "server" instance we must also set the values for the two leafs inside the server instance, the `ip` and the `port` elements. If we fail to do this, commit will fail.

There are two variants on the MAAPI function which sets a a value. One where the value is a regular string, and one where the value is of type confd_value_t. Usually when we implement a proprietary agent, somehow a user will enter values as strings. In our code above, we use the string variant.

This program can be found in the examples section of a ConfD release, in the `misc/maapi_cli` directory.

If we want to use MAAPI to implement a general purpose agent which works on all data models, we can use the `confd_cs_node` tree that is generated when we call `maapi_load_schemas()`/ `confd_load_schemas()`, see the section called “USING SCHEMA INFORMATION” in the confd_types(3) manual page. This is a representation of the complete data model in the form of a tree of linked C structures.

Chapter 23. High Availability

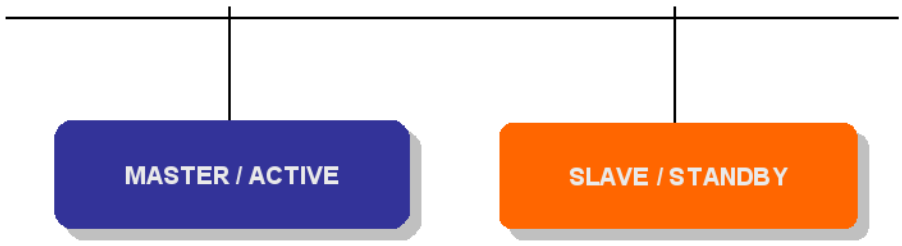
23.1. Introduction to ConfD High Availability

ConfD support replication of the CDB configuration as well as of the operational data kept in CDB. The replication architecture is that of one active master and a number of passive slaves.

All configuration write operations must occur at the master and ConfD will automatically distribute the configuration updates to the set of live slaves. 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 also occur at the master, with the updates distributed to the live slaves, whereas non-replicated operational data can also be written on the slaves.

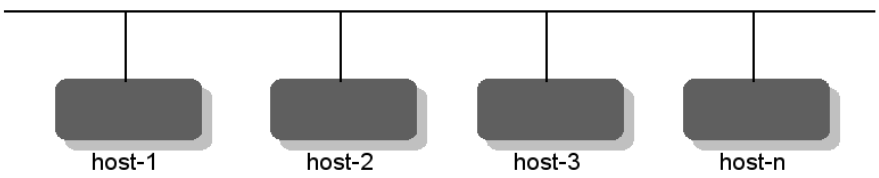
The *only* thing ConfD does is to replicate the CDB data amongst the members in the HA group. It doesn't perform any of the otherwise High-Availability related tasks such as running election protocols in order to elect a new master. This is the task of a High-Availability Framework (HAFW) which must be in place. The HAFW must instruct ConfD which nodes are up and down using API functions from confd_lib_ha(3). Thus in order to use ConfD configuration replication we must first have a HAFW.

Replication is supported in several different architectural setups. For example two-node active/standby designs as well as multi-node clusters with runtime software upgrade.



Master - Slave configuration

In a chassis solution there are (at least two) but a fixed number of management blades. We wish that all configuration data is replicated and if the active dies the standby will takeover and the configuration data is not lost.



One master - several slaves

Furthermore it is assumed that the entire cluster configuration is equal on all hosts in the cluster. This means that node specific configuration must be kept in different node specific subtrees, for example as in Example 23.1, “A data model divided into common and node specific subtrees”.

Example 23.1. A data model divided into common and node specific subtrees

```
container cfg {
  container shared {
    leaf dnsserver {
      type inet:ipv4-address;
    }
    leaf defgw {
      type inet:ipv4-address;
    }
    leaf token {
      type AESCFB128EncryptedString;
    }
    ...
  }
  container cluster {
    list host {
      key ip;
      max-elements 8;
      leaf ip {
        type inet:ipv4-address;
      }
     ...
    }
  }
}
```

23.2. HA framework requirements

ConfD only replicates the CDB data. ConfD must be told by the HAFW which node should be master and which nodes should be slaves.

The HA framework must also detect when nodes fail and instruct ConfD accordingly. If the master node fails, the HAFW must elect one of the remaining slaves and appoint it the new master. The remaining slaves must also be informed by the HAFW about the new master situation. ConfD will never take any actions regarding master/slave-ness by itself.

23.3. Mode of operation

ConfD must be instructed through the `confd.conf` configuration file that it should run in HA mode. The following configuration snippet enables HA mode:

```
<ha>
  <enabled>true</enabled>
  <ip>0.0.0.0</ip>
  <port>4569</port>
  <tickTimeout>PT20S</tickTimeout>
</ha>
```

The IP address and the port above indicates which IP and which port should be used for the communication between the HA nodes. The `tickTimeout` is a duration indicating how often each slave must send a tick

message to the master indicating liveness. If the master has not received a tick from a slave within 3 times the configured tick time, the slave is considered to be dead. Similarly, the master sends tick messages to all the slaves. If a slave has not received any tick messages from the master within the 3 times the timeout, the slave will consider the master dead and report accordingly.

A HA node can be in one of three states: `NONE`, `SLAVE` or `MASTER`. Initially a node is in the `NONE` state. This implies that the node will read its configuration from CDB, stored locally on file. Once the HA framework has decided whether the node should be a slave or a master the HAFW must invoke either the function `confd_ha_beslave(master)` or `confd_ha_bemaster()`.

When a ConfD HA node starts, it always starts up in mode `NONE`. This is consistent with how ConfD works without HA enabled. At this point there are no other nodes connected. Each ConfD node reads its configuration data from the locally stored CDB and applications on or off the node may connect to ConfD and read the data they need.

At some point, the HAFW will command some nodes to become slave nodes of a named master node. When this happens, each slave node tracks changes and (logically or physically) copies all the data from the master. Previous data at the slave node is overwritten.

Note that the HAFW, by using ConfD's start phases, can make sure that ConfD does not start its northbound interfaces (NETCONF, CLI, ...) until the HAFW has decided what type of node it is. Furthermore once a node has been set to the SLAVE state, it is not possible to initiate new write transactions towards the node. It is thus never possible for an agent to write directly into a slave node. Once a node is returned either to the NONE state or to the MASTER state, write transactions can once again be initiated towards the node.

The HAFW may command a slave node to become master at any time. The slave node already has up-to-date data, so it simply stops receiving updates from the previous master. Presumably, the HAFW also commands the master node to become a slave node, or takes it down or handles the situation somehow. If it has crashed, the HAFW tells the slave to become master, restarts the necessary services on the previous master node and gives it an appropriate role, such as slave. This is outside the scope of ConfD.

Each of the master and slave nodes have the same set of all callpoints and validation points locally on each node. The start sequence has to make sure the corresponding daemons are started before the HAFW starts directing slave nodes to the master, and before replication starts. The associated callbacks will however only be executed at the master. If e.g. the validation executing at the master needs to read data which is not stored in the configuration and only available on another node, the validation code must perform any needed RPC calls.

If the order from the HAFW is to become master, the node will start to listen for incoming slaves at the `ip:port` configured under `/confdCfg/ha`. The slaves TCP connect to the master and this socket is used by ConfD to distribute the replicated data.

If the order is to be a slave, the node will contact the master and possibly copy the entire configuration from the master. This copy is not performed if the master and slave decide that they have the same version of the CDB database loaded, in which case nothing needs to be copied. This mechanism is implemented by use of a unique token, the "transaction id" - it contains the node id of the node that generated it and and a time stamp, but is effectively "opaque".

This transaction id is generated by the cluster master each time a configuration change is committed, and all nodes write the same transaction id into their copy of the committed configuration. If the master dies, and one of the remaining slaves is appointed new master, the other slaves must be told to connect to the new master. They will compare their last transaction id to the one from the newly appointed master. If they are the same, no CDB copy occurs. This will be the case unless a configuration change has sneaked in, since both the new master and the remaining slaves will still have the last transaction id generated by the old master - the new master will not generate a new transaction id until a new configuration change

is committed. The same mechanism works if a slave node is simply restarted. In fact no cluster reconfiguration will lead to a CDB copy unless the configuration has been changed in between.

Northbound agents should run on the master, it is not possible for an agent to commit write operations at a slave node.

When an agent commits its CDB data, CDB will stream the committed data out to all registered slaves. If a slave dies during the commit, nothing will happen, the commit will succeed anyway. When and if the slave reconnects to the cluster, the slave will have to copy the entire configuration. All data on the HA sockets between ConfD nodes only go in the direction from the master to the slaves. A slave which isn't reading its data will eventually lead to a situation with full TCP buffers at the master. In principle it is the responsibility of HAFW to discover this situation and notify the master ConfD about the hanging slave. However if 3 times the tick timeout is exceeded, ConfD will itself consider the node dead and notify the HAFW. The default value for tick timeout is 20 seconds.

The master node holds the active copy of the entire configuration data in CDB. All configuration data has to be stored in CDB for replication to work. At a slave node, any request to read will be serviced while write requests will be refused. Thus, CDB subscription code works the same regardless of whether the CDB client is running at the master or at any of the slaves. Once a slave has received the updates associated to a commit at the master, all CDB subscribers at the slave will be duly notified about any changes using the normal CDB subscription mechanism.

23.4. Security aspects

We specify in `confd.conf` which IP address the master should bind for incoming slaves. If we choose the default value `0.0.0.0` it is the responsibility of the application to ensure that connection requests only arrive from acceptable trusted sources through some means of firewalling.

A cluster is also protected by a token, a secret string only known to the application. The API function `confd_ha_connect()` must be given the token. A slave node that connects to a master node negotiates with the master using a CHAP-2 like protocol, thus both the master and the slave are ensured that the other end has the same token without ever revealing their own token. The token is never sent in clear text over the network. This mechanism ensures that a connection from a ConfD slave to a master can only succeed if they both have the same token.

It is indeed possible to store the token itself in CDB, thus an application can initially read the token from the local CDB data, and then use that token in `confd_ha_connect()`. In this case it may very well be a good idea to have the token stored in CDB be of type tailf:aes-cfb-128-encrypted-string.

If the actual CDB data that is sent on the wire between cluster nodes is sensitive, and the network is untrusted, the recommendation is to use IPSec between the nodes. An alternative option is to decide exactly which configuration data is sensitive and then use the tailf:aes-cfb-128-encrypted-string type for that data. If the configuration data is of type tailf:aes-cfb-128-encrypted-string the encrypted data will be sent on the wire in update messages from the master to the slaves.

23.5. API

There are two APIs used by the HA framework to control the replication aspects of CDB. First there exists a synchronous API used to tell ConfD what to do, secondly the application may create a notifications socket and subscribe to HA related events where ConfD notifies the application on certain HA related events such as the loss of the master etc. This notifications API is described in ????. The HA related notifications sent by ConfD are crucial to how to program the HA framework.

The following functions are used from the HAFW to instruct ConfD about the cluster.

```
int confd_ha_connect(int sock, const struct sockaddr* srv, int
srv_sz, const char *token);
```

Connects a HA socket to ConfD and also provides the secret token to be used in later negotiations with other nodes.

```
int confd_ha_bemaster(int sock, confd_value_t *mynodeid);
```

Instructs the local node to become master. The function also provides a node identifier for the node. The node id is of type confd_value_t. Thus if we in our configuration have trees with different branches for node local data, it is highly recommended to use the same type there as for the type of the node id.

```
int confd_ha_beslave(int sock, confd_value_t *mynodeid, struct
confd_ha_node *master, int waitreply);
```

Instructs a node to be slave. The definition of the struct confd_ha_node is:

```
struct confd_ha_node {
    confd_value_t nodeid;
    int af;               /* AF_INET | AF_INET6 | AF_UNSPEC */
    union {               /* address of remote note */
        struct in_addr ip4;
        struct in6_addr ip6;
        char *str;
    } addr;
    char buf[128];        /* when confd_read_notification() and        */
                          /* confd_ha_status() populate these structs, */
                          /* if type of nodeid is C_BUF, the pointer   */
                          /* will be set to point into this buffer     */
    char addr_buf[128];   /* similar to the above, but for the address */
                          /* of remote node when using external IPC    */
                          /* (from getpeeraddr() callback for slaves)  */
};
```

```
int confd_ha_benone(int sock);
```

Resets a node to the initial state.

```
int confd_ha_berelay(int sock);
```

Instructs a slave node to be a relay for other slaves. This is discussed in Section 23.8, “Relay slaves”.

```
int confd_ha_status(int sock, struct confd_ha_status *stat);
```

Returns the status of the current node in the user provided struct confd_ha_status structure. The definition is:

```
struct confd_ha_status {
    enum confd_ha_status_state state;
    /* if state is MASTER, we also have a list of slaves */
    /* if state is SLAVE, then nodes[0] contains the master */
    /* if state is RELAY_SLAVE, then nodes[0] contains the master,
       and following entries contain the "sub-slaves" */
    /* if state is NONE, we have no nodes at all */
```

```
    struct confd_ha_node nodes[255];
    int num_nodes;
};
```

```
int confd_ha_slave_dead(int sock, confd_value_t *nodeid);
```

This function must be used by the HAFW to tell ConfD that a slave node is dead. It is vital that this is indeed executed. ConfD will notice that a slave is dead automatically if the socket to the slave is closed, however the slave can die without closing its socket. If configured, ConfD will periodically send alive tick messages from the slaves to the master. If a tick message isn't received by the master within the pre configured time the master will consider the slave dead, close the socket and report to the application through a notifications socket.

23.6. Ticks

The configuration parameter `/confdCfg/ha/tickTimeout` is by default set to 20 seconds. This means that every 20 seconds each slave will send a tick message on the socket leading to the master. Similarly, the master will send a tick message every 20 seconds on every slave socket.

This aliveness detection mechanism is necessary for ConfD. If a socket gets closed all is well, ConfD will cleanup and notify the application accordingly using the notifications API. However, if a remote node freezes, the socket will not get properly closed at the other end. ConfD will distribute update data from the master to the slaves. If a remote node is not reading the data, TCP buffer will get full and ConfD will have to start to buffer the data. ConfD will buffer data for at most `tickTime` times 3 time units. If a `tick` has not been received from a remote node within that time, the node will be considered dead. ConfD will report accordingly over the notifications socket and either remove the hanging slave or, if it is a slave that loose contact with the master, go into the initial `NONE` state.

If the HAFW can be really trusted, it is possible to set this timeout to `PT0S`, i.e zero, in which case the entire dead-node-detection mechanism in ConfD is disabled.

23.7. Joining a cluster

Some applications consist of several machines and also have an architecture where it is possible to dynamically add more machines to the cluster. The procedure to add a machine to the cluster is called “joining the cluster”.

Assume a situation where the cluster is running, we know that the master is running at IP address `master_ip`. A common technique is to bring up a virtual IP address (VIP) on the master and then use gratuitous ARP to inform the other hosts on the same L2 network about the new MAC/IP mapping.

The code to join a cluster is always going to be application specific. Typically we would do something like the following:

1. Start the new machine with an initial simple CLI which gathers the following information from the user or from the network.

   - The VIP. We need to know where the master is. This can be entered manually. Another technique would be to use UDP broadcast at the new machine and let code running at the master reply. Regardless, we need an IP address to connect to.

   - The admin password.

2. Connect to a server at the VIP and send the admin password. This server code must then:

   - Use `maapi_authenticate()` to check that the remote user indeed knows the admin password (or whichever user we choose in our application).

   - Assume a data model similar to the one in Example 23.1, “A data model divided into common and node specific subtrees”. The server code running at the master would then use MAAPI to populate the new `/cfg/cluster/host` tree for the joining slave. Finally the master code replies with the secret cluster token found in the master config at `/cfg/shared/token`. It is not necessary to have the token in CDB, it could also be stored somewhere else or even hard coded if the network for cluster communication is considered trusted.

   - The join code at the new machine now has the token. It can start ConfD with its default configuration. Once ConfD is started the join code invokes `confd_ha_beslave()` and we are done.

23.8. Relay slaves

The normal setup of a ConfD HA cluster is to have all slaves connected directly to the master. This is a configuration that is both conceptually simple and reasonably straightforward to manage for the HAFW. In some scenarios, in particular a cluster with multiple slaves at a location that is network-wise distant from the master, it can however be sub-optimal, since the replicated data will be sent to each remote slave individually over a potentially low-bandwidth network connection.

To make this case more efficient, we can instruct a slave to be a relay for other slaves, by invoking the `confd_ha_berelay()` API function. This will make the slave start listening on the IP address and port configured for HA in `confd.conf`, and handle connections from other slaves in the same manner as the cluster master does. The initial CDB copy (if needed) to a new slave will be done from the relay slave, and when the relay slave receives CDB data for replication from its master, it will distribute the data to all its connected slaves in addition to updating its own CDB copy.

To instruct a node to become a slave connected to a relay slave, we use the `confd_ha_beslave()` function as usual, but pass the node information for the relay slave instead of the node information for the master. I.e. the "sub-slave" will in effect consider the relay slave as its master. To instruct a relay slave to stop being a relay, we can invoke the `confd_ha_beslave()` function with the same parameters as in the original call. This is a no-op for a "normal" slave, but it will cause a relay slave to stop listening for slave connections, and disconnect any already connected "sub-slaves".

This setup requires special consideration by the HAFW. Instead of just telling each slave to connect to the master independently, it must setup the slaves that are intended to be relays, and tell them to become relays, before telling the "sub-slaves" to connect to the relay slaves. Consider the case of a master M and a slave S0 in one location, and two slaves S1 and S2 in a remote location, where we want S1 to act as relay for S2. The setup of the cluster then needs to follow this procedure:

1. Tell M to be master.

2. Tell S0 and S1 to be slave with M as master.

3. Tell S1 to be relay.

4. Tell S2 to be slave with S1 as master.

Conversely, the handling of network outages and node failures must also take the relay slave setup into account. For example, if a relay slave loses contact with its master, it will transition to the `NONE` state just like any other slave, and it will then disconnect its "sub-slaves" which will cause those to transition to `NONE` too, since they lost contact with "their" master. Or if a relay slave dies in a way that is detected

by its "sub-slaves", they will also transition to `NONE`. Thus in the example above, S1 and S2 needs to be handled differently. E.g. if S2 dies, the HAFW probably won't take any action, but if S1 dies, it makes sense to instruct S2 to be a slave of M instead (and when S1 comes back, perhaps tell S2 to be a relay and S1 to be a slave of S2).

Besides the use of `confd_ha_berelay()`, the API is mostly unchanged when using relay slaves. The HA event notifications reporting the arrival or the death of a slave are still generated only by the "real" cluster master. If the `confd_ha_status()` API function is used towards a relay slave, it will report the node state as CONFD_HA_STATE_SLAVE_RELAY rather than just CONFD_HA_STATE_SLAVE, and the array of nodes will have its master as the first element (same as for a "normal" slave), followed by its "sub-slaves" (if any).

23.9. CDB replication

When HA is enabled in `confd.conf` CDB automatically replicates data written on the master to the connected slave nodes. Replication is done on a per-transaction basis to all the slaves in parallel. It can be configured to be done asynchronously (best performance) or synchronously in step with the transaction (most secure). When ConfD is in slave mode the northbound APIs are in read-only mode, that is the configuration can not be changed on a slave other than through replication updates from the master. It is still possible to read from for example NETCONF or CLI (if they are enabled) on a slave. CDB subscriptions works as usual. When ConfD is in the NONE state CDB is unlocked and it behaves as when ConfD is not in HA mode at all.

The Section 6.8, “Operational data in CDB” describes how operational data can be stored in CDB. If this is used it is also possible to replicate operational data in HA mode. Since replication comes at a cost ConfD makes it configurable whether to replicate all operational data, or just the persistent data (the default). See the confd.conf(5) man-page for the `/confdConfig/cdb/operational/replication` configurable. Replication of operational data can also be configured to be done asynchronously or synchronously, via the `/confdConfig/cdb/operational/replicationMode` configurable, but since there are no transactions for the writing of operational data, this pertains to a given API call writing operational data.

Chapter 24. The SNMP Gateway

24.1. Introduction to the ConfD SNMP Gateway

By using the SNMP gateway, ConfD makes SNMP data available through the management interfaces (such as CLI and NETCONF). The idea is that ConfD can co-exist with external SNMP agents on the device, and use SNMP (in the simplest case it will be SNMP over the loopback interface) to retrieve data from the agents, and present it over e.g. NETCONF.

What is needed to access the data provided by an SNMP agent is the MIB files defining the data. The MIB modules are translated into read-only YANG modules, using the standard mapping defined in http://www.ietf.org/rfc/rfc6643.txt. After compiling the YANG files, as described in earlier chapters, ConfD can load the resulting `.fxs` files and can then provide data from the SNMP agent through the various ConfD interfaces (CLI, NETCONF, etc.).

The gateway supports SNMP v1 and v2c when it communicates with the SNMP agent. SNMP v2c is preferred over v1, since it is more efficient.

24.2. Configuring Agent Access

In the ConfD configuration file `confd.conf`, the location of SNMP agents and characteristics of the communication with them can be specified with the `/confdConfig/snmpgw` element. An example is shown below:

Example 24.1. Example snmpgw configuration fragment in confd.conf

```
<snmpgw>
  <enabled>true</enabled>
  <trapPort>5000</trapPort>
  <agent>
    <name>a1</name>
    <subscriptionId>id1</subscriptionId>
    <enabled>true</enabled>
    <community>private</community>
    <version>v2c</version>
    <timeout>PT2S</timeout>
    <ip>127.0.0.1</ip>
    <port>161</port>
    <module>ONE-MIB</module>
    <module>TWO-MIB</module>
  </agent>
  <agent>
    <name>a2</name>
    <subscriptionId>id2</subscriptionId>
    <enabled>true</enabled>
    <community>private</community>
    <version>v2c</version>
    <timeout>PT2S</timeout>
    <ip>192.168.1.12</ip>
    <port>161</port>
    <module>THIRD-MIB</module>
  </agent>
</snmpgw>
```

Each `/confdConfig/snmpgw/agent` element is called an SNMP agent configuration element. It has to have a unique `name` (mainly for error reporting), and relates a subset of the configuration to a particular SNMP agent. The element `module`, which can be present multiple times, specifies which MIBs the agent implements. Each such MIB must be converted to YANG and compiled into an `.fxs` file.

The `subscriptionId` and `trapPort` elements are described in the section *Receiving and Forwarding Notifications*.

The default value for the `enabled` element is `true`. If the value is `false`, this agent element is disregarded.

The possible values for the SNMP protocol version are `v1` and `v2c`. `v2c` is preferred.

The `/confdConfig/snmpgw/agent/timeout` element has the type xs:duration. Timeout when communicating with the SNMP agent produces an error, and the ConfD operation is aborted.

In addition to the `community` element, which only allows for the specification of Unicode community strings, the element `community_bin` can be used for specifying arbitrary community strings, in the hexadecimal format xs:hexBinary. For example, `<community_bin>004103</community_bin>` specifies a string with three bytes; 0x00, 0x41 and 0x03. If both `community_bin` and `community` are given, the latter is ignored.

24.3. Compiling the MIBs

Each MIB is converted to YANG using **confdc** with the parameter `--mib2yang-std`. Then the resulting YANG module is compiled using **confdc** with the parameters `-c --snmpgw`.

```
$ confdc --mib2yang-std -o IF-MIB.yang IF-MIB.mib
$ confdc -c --snmpgw -o IF-MIB.fxs IF-MIB.yang
```

24.4. Receiving and Forwarding Notifications

For the purpose of forwarding SNMP notifications (also called traps) from external agents to a user application, the SNMP gateway can be made to listen for notifications on the port indicated by the element `/confdConfig/snmpgw/trapPort`. If this element is not present, listening for notifications is disabled.

Only SNMPv2 notifications are handled.

Each agent configuration element has a child element `subscriptionId` (of type xs:token). When a notification arrives, its sender (IP address and port) is compared with the agent configuration elements, and a subscriptionId determined. If any application has subscribed to that subscription id, the notification is sent to it, otherwise it is dropped. A given ip/port pair should only be handled by one subscriptionId, otherwise the `confd.conf` file is rejected.

To register its interest in notification reception, the application should call the function `confd_register_notification_sub_snmp_cb()`. Example:

Example 24.2. C code for registering reception of notifications

```
int recv_snmp(struct confd_notification_ctx *nctx, char *notif,
              struct confd_snmp_varbind *vb, int n,
              confd_value_t *srcaddr, u_int16_t srcport)
{
```

```
    ...
}

int main()
{
    ...

    struct confd_daemon_ctx *dctx;
    struct confd_notification_sub_snmp_cb snmpcb;

    dctx = confd_init_daemon(dname, debuglevel, stderr);
    [ connect control and worker sockets ]
    strcpy(snmpcb.sub_id, "id1");
    snmpcb.recv = recv_snmp;
    confd_register_notification_sub_snmp_cb(dctx, &snmpcb);
    confd_register_done(dctx);

    ...
}
```

"id1" here is the subscription id. The function `recv_snmp()` will be called when a notification arrives:

`notif` is the name of the notification, if it can be obtained from the notification id (if the relevant MIB is loaded into the agent), otherwise the empty string. `srcaddr`/`srcport` are the IP address and port of the notification's (immediate) sender. The notification id appears in second position among the variables, and a timestamp in first position, as they should in a well-formed notification.

One thing an application may want to do with a received notification is to forward it somewhere else. See Section 17.2.15, “Notifications” for how to do that.

24.5. Example Scenario

In the following example, we assume that there exist a MIB `OUR-MIB.mib` and we wish to translate into a ConfD `.fxs` file so that we can use ConfD to access the MIB data. The example is very small; real-life MIBs are likely to also depend on several standard MIBs.

The following steps produce the file that ConfD needs (namely, `OUR-MIB.fxs`):

Example 24.3. Example 1 of translating and compiling a MIB

```
$ confdc --mib2yang-std -o OUR-MIB.yang OUR-MIB.mib
$ confdc -c --snmpgw OUR-MIB.yang
```

Chapter 25. Subagents and Proxies

25.1. Introduction

ConfD supports a master/subagent concept similar to that found in e.g. AgentX (RFC 2741). The idea is that there is one master agent running on a managed device. It terminates the northbound interfaces such as NETCONF and CLI. The master agent is connected to a set of subagents which provide instrumentation of the subsystems.

A subagent has its own data store, separate from the master agent. A subagent is an essential part of the system, i.e. if the master agent cannot talk to the subagent, this is handled as a data provider failure.

Subagents may be used in a chassis based system when some of the blades may also ship as standalone products. In this case it is desirable to have identical software on the blade regardless of weather the blade sits in a chassis or is shipped as a standalone product.

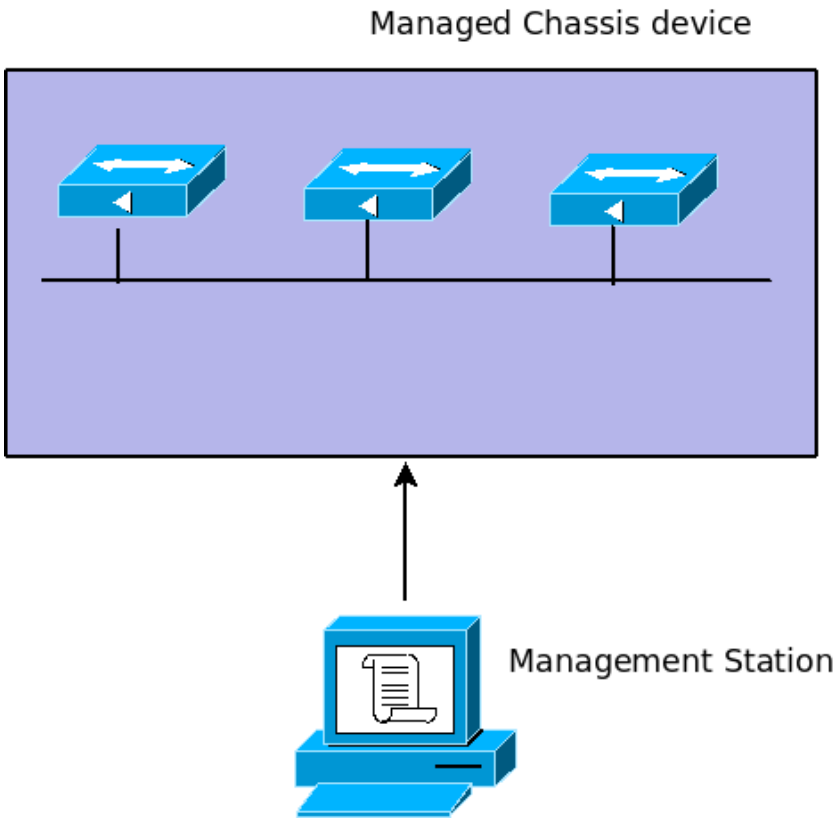
Subagents are also the right choice when there is a need to to integrate software that already has a management interface of its own. In this case, it is desirable not to change that code, but still make it appear as an integrated part of the entire chassis. A typical usage scenario is when there is an existing standalone product that also should be part of a chassis solution.

Subagents are not the right choice for supporting field replaceable units (FRU), such as interface cards. In this case, it is recommended to have the software on the FRU connect to a ConfD running on a management processor through the normal ConfD C-APIs.

In ConfD, NETCONF is used as master-to-subagent protocol. The subagent only has to provide a NETCONF interface. The master agent can provide any northbound interfaces, for example CLI and Web UI only. This is accomplished in ConfD by separating the northbound agents from the data providers. Somewhat simplified, the subagents are viewed and handled as any other data provider.

Authentication and authorization (access control) is done by the master agent. This means that access control rules are configured at the master agent, and checked in runtime at the master. The subagent should be configured to allow full access to the user which the master agent uses for the connections.

The following picture illustrates how a chassis based system internally consists of three different subsystems.



Multiple devices within a chassis system

Subagents are used when management station should perceive the system as a whole; thus subagents can be viewed as an internal implementation detail, not visible from the outside.

Another common architecture is that of configuring one ConfD instance to be able to proxy configuration traffic explicitly to one or more other managed devices.

This architecture is usable in similar situations as subagents with the exception that the outside management station must have explicit knowledge about the internal subsystems. It is common to build chassis based systems that consist of several subsystems that are themselves devices. The internal devices are typically not reachable on the external network, they are attached to network internal to the chassis - thus the need for a proxy solution.

25.2. Subagent Registration

The subagents are registered at the master agent. Information about each subagent is written into the master agent's `confd.conf` file. The subagent configuration is marked as "reloadable", see Section 27.4, “Configuring ConfD”, so it is possible for the application to easily use parts of the subagent configuration in its own configuration.

Once a subagent is enabled, it will be viewed as an essential part of the system, i.e. if the master agent cannot talk to the subagent, this is handled as a data provider failure. This means that operations like <edit-config> will fail if the master cannot contact an enabled subagent. ConfD sends an event to the application when it detects communication failures with subagents. The event is described in the confd_lib_events(3) man page. The application can choose to disable or remove the subagent if it wants to, either by modifying the master agent's `confd.conf` file and do **confd --reload**, or by directly changing the configuration parameters (see Section 27.4, “Configuring ConfD”).

The registration information needed per subagent is:

subagent address and transport
: Currently supported transports are SSH and TCP. TCP is non-standard, but unencrypted and thus more efficient.

authentication information
: For SSH, specify the username and password that ConfD will use when connecting to the subagent.

  For TCP, the ConfD specific TCP header described in the NETCONF chapter is used. This means that the user name and groups have to defined for the subagent.

registration path
: An XPath expression which defines where in the master agent's data hierarchy the subagent's data is registered. For example `/config/blade[id="3"]` `/config/ospf`, or just `/`.

Each subagent registers a set of top-elements from one or more namespaces. These nodes will be mounted at the registration path at the master agent.

The data model that the subagent registers must be available at the master agent, in the form of a .fxs file in the normal load path. This .fxs file must be compiled with the flag `--subagent MountPath` before it is loaded in the master agent. This option tells the master agent that this namespace is handled by a subagent. `MountPath` is the same as the registration path in `confd.conf`, but without any instance selectors.

25.2.1. Example

Here is a step-by-step example on how to add three subagents, called A, B and C, to a master agent. We will assume that A and B implement one instance each of some service. A implements the SMTP service, and B IMAP and POP. Subagent C implements the equipment subsystem. The idea is that there might be more than one SMTP service or IMAP service, but a single equipment subsystem.

If a client talks directly to A, it will get the following data:

Example 25.1. smtp subagent data

```
<smtp-config xmlns="http://example.com/smtp/1.0">
  <enabled>true</enabled>
  ...
</smtp-config>
```

If a client talks directly to B, it will get the following data:

Example 25.2. imap and pop subagent data

```
<imap-config xmlns="http://example.com/imap/2.1">
  <enabled>true</enabled>
  ...
</imap-config>
<pop-config xmlns="http://example.com/pop/1.2">
  <enabled>true</enabled>
  ...
</pop-config>
```

If a client talks directly to C, it will get the following data:

Example 25.3. Equipment subagent data

```
<config xmlns="http://example.com/equipment/2.1">
  <chassis>
    ...
  </chassis>
</config>
```

At the master agent, we want the following data:

Example 25.4. master agent data

```
<system xmlns="http://example.com/service/3.3">
  <services>
    <service>
      <name>smtp1</name>
      <type>smtp</type>
      <smtp-config xmlns="http://example.com/smtp/1.0">
        <enabled>true</enabled>
        ...
      </smtp-config>
    </service>
    <service>
      <name>imap1</name>
      <type>imap</type>
      <imap-config xmlns="http://example.com/imap/2.1">
        <enabled>true</enabled>
        ...
      </imap-config>
    <service>
      <name>pop1</name>
      <type>pop</type>
      <pop-config xmlns="http://example.com/pop/1.2">
        <enabled>true</enabled>
         ...
        </pop-config>
      </pop-config>
    </service>
  </services>
</system>
<config xmlns="http://example.com/equipment/2.1">
  <chassis>
    ...
  </chassis>
</config>
```

The first thing to do at the master agent is to compile the YANG modules:

Example 25.5. Compile the YANG modules at the master

```
$ confdc -c --subagent /system/services/service -o smtp.fxs smtp.yang

$ confdc -c --subagent /system/services/service -o imap.fxs imap.yang

$ confdc -c --subagent /system/services/service -o pop.fxs pop.yang

$ confdc -c --subagent / -o equip.fxs equip.yang
```

Next, we put the following into `confd.conf`:

Example 25.6. Master agent's confd.conf

```
<subagents>

  <enabled>true</enabled>

  <subagent>
    <name>A</name>
    <enabled>true</enabled>
    <tcp>
      <ip>10.0.0.1</ip>
      <port>2023</port>
      <confdAuth>
        <user>admin</user>
        <group>admin</group>
      </confdAuth>
    </tcp>
    <mount xmlns:sa="http://example.com/smtp/1.0">
      <path>/system/services/service[name="smtp1"]</path>
      <node>sa:smtp-config</node>
    </mount>
  </subagent>

  <subagent>
    <name>B</name>
    <enabled>true</enabled>
    <tcp>
      <ip>10.0.0.2</ip>
      <port>2023</port>
      <confdAuth>
        <user>admin</user>
        <group>admin</group>
      </confdAuth>
    </tcp>
    <mount xmlns:imap="http://example.com/imap/2.1"
           xmlns:pop="http://example.com/pop/1.3">
      <path>/system/services/service[name="imap1"]
            /system/services/service[name="pop1"]</path>
      <node>imap:imap-config pop:pop-config</node>
    </mount>
  </subagent>

  <subagent>
    <name>C</name>
    <enabled>true</enabled>
    <tcp>
      <ip>127.0.0.1</ip>
      <port>2043</port>
      <confdAuth>
        <user>admin</user>
        <group>admin</group>
      </confdAuth>
    </tcp>
    <mount xmlns:sa="http://example.com/equipment/2.1">
      <path>/</path>
      <node>sa:config</node>
```

```
    </mount>
  </subagent>
```

Note that the instances `/services/service[name="smtp1"]`, `/services/service[name="imap1"]`, and `/services/service[name="pop1"]` must be created in the database at the master agent before the subagent will be used.

25.3. Subagent Requirements

Some of the capabilities the master agent advertises must be supported among all subagents. For example, in order for the master agent to advertise the startup capability, all subagents must support it. Some other capabilities can be handled entirely in the master agent, and can be advertised independently of the subagents.

`:writable-running`, *`:startup`*, *`:confirmed-commit`*, *`:validate`*
: These capabilities can be advertised by the master agent if all subagents support them.

`:candidate`
: This capabilities can be advertised by the master agent if all subagents support them. In this case, the master ConfD must be configured with `/confdConfig/datastores/candidate/implementation` set to `external` in confd.conf.

`:rollback-on-error`
: This capability can be advertised by the master agent if all subagents support the `http://tail-f.com/ns/netconf/transactions/1.0` capability. One exception to this is if there is one single subagent which doesn't support the 'transactions' capability (and zero or more agents supporting it), and this single agent supports *`:rollback-on-error`*. For more information on the 'transactions' capability, see Section 15.10, “Transactions Capability”.

`:xpath`
: This capability can be advertised by the master agent independently of the subagents. The subagents do not have to support XPath.

`:url`
: This capability can be advertised by the master agent independently of the subagents. The subagents do not have to support the *`:url`* capability.

25.4. Proxies

ConfD can be configured to proxy NETCONF traffic and CLI sessions. The configuration of the proxies reside in confd.conf. The proxy configuration is marked as "reloadable", see Section 27.4, “Configuring ConfD”, so it is possible for the application to easily use parts of the proxy configuration in its own configuration.

As an example, assume we have a chassis system with two internal boards that reside on a chassis internal network that is not reachable from the outside. We still want the operators to be able to configure the boards, thus we instruct ConfD to proxy network traffic to the internal boards. An example configuration snippet (from confd.conf) could be:

Example 25.7. Proxy configuration

```
<proxyForwarding>
  <enabled>true</enabled>
```

```
  <autoLogin>true</autoLogin>
  <proxy>
    <target>board-1</target>
    <address>10.10.0.1</address>
    <netconf>
      <ssh>
        <port>830</port>
      </ssh>
    </netconf>
    <cli>
      <ssh>
        <port>22</port>
      </ssh>
    </cli>
  </proxy>

  <proxy>
    <target>board-2</target>
    <address>10.10.0.2</address>
    <netconf>
      <ssh>
        <port>830</port>
      </ssh>
    </netconf>
    <cli>
      <ssh>
        <port>22</port>
      </ssh>
    </cli>
  </proxy>
</proxyForwarding>

<netconf>
  <capabilities>
    <forward>
      <enabled>true</enabled>
    </forward>
    <!-- other capabilities here ... -->
  </capabilities>
  <!-- more netconf config here ... -->
</netconf>
```

The above instructs ConfD to proxy forward CLI traffic and NETCONF traffic from the "Management interface host" (MIH) to the "Internal hosts" (IH) Both types of traffic must be explicitly initiated by the operator.

We define two internal hosts to which we wish to proxy traffic. Each internal host has a symbolic name which is is used by both the CLI operator as well as the NETCONF client.

For all internal hosts we define weather we want to attempt auto login or not. If the ConfD internal SSH server was used in the original connection to the management interface host, be it NETCONF or CLI, ConfD has access to the clear text password. In that case an SSH connection attempt will be made with the same username/password pair as the original connection. If that fails, the NETCONF session will fail with a error whereas the CLI will prompt for a new password. If ConfD does not have access to the SSH password for the original connection to the management interface host, a password must be explicitly supplied by the CLI operator/NETCONF client.

It is of course also possible to arrange private/public keys on the chassis host in such a manner so that passwords will never be used.

25.4.1. CLI forwarding

The CLI user must explicitly initiate SSH connections to the internal hosts using the builtin "forward" command in the CLI. The single argument of the "forward" command is the string defined as "target" in confd.conf. The SSH connection to the target will be made with the same userid as the original CLI connection has.

```
admin@chassis> forward [TAB]
Possible completions:
  board-1 - 10.10.0.1:22
  board-2 - 10.10.0.2:22
admin@chassis> forward board-1
admin@board-1> id
user = admin(2), gid=3, groups=admin, gids=
[ok][2008-08-15 12:14:41]
admin@board-1> ^D
Connection to board-1 closed
[ok][2008-08-15 12:14:58]
admin@chassis>
```

The above (Juniper style CLI) shows a session where the CLI operator connects the CLI to an internal host (board-1)

25.4.2. NETCONF forwarding

ConfD publishes a new "proxy forwarding" NETCONF capability. If the management station issues the forward command, ConfD relays this connection to the IH. The proxy forwarding capability is defined in the NETCONF chapter.

If the command succeeds, any messages arriving in this session would subsequently be forwarded to the target device without any analysis on the forwarding device. This channel is also open to NETCONF notifications sent from the IH. This goes on until the session is closed.

A NETCONF session that connects to board-1 and asks for the dhcp configuration could like this:

Example 25.8. Agent replies with forward capability

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>http://tail-f.com/ns/netconf/forward/1.0</capability>
  </capabilities>
</hello>
```

Example 25.9. Manager issues forward rpc to board-1

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>board-1</target>
  </forward>
</rpc>
```

Example 25.10. Manager issues command

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">
  <get>
```

```
    <filter>
      <dhcp xmlns="http://tail-f.com/ns/example/dhcpd/1.0"/>
    </filter>
  </get>
</rpc>
```

This last `get` request will be forwarded to the IH by the MIH. Finally the manager issues a `close-session` request whereby the manager will have the original SSH connection back to the IMH.

Example 25.11. close-session

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="3">
  <close-session/>
</rpc>
```

When ConfD at the MIH sees the "forward" command, ConfD looks up the IH identity in its configuration which provides a mapping to the appropriate IP address. ConfD then establishes an SSH connection to the IH.

The "forward" command may require authentication from the user. This happens if ConfD is not configured to do automatic login to the IH, or if automatic login fails. In this case, the reply will be 'not-authorized'.

The authentication protocol is SASL (RFC 4422), using the XML mapping defined for XMMP (RFC 3920). ConfD supports the PLAIN authentication mechanism (RFC 4616).

On successful completion of the "forward" command, the IH's capabilities are returned in the "rpc-reply".

When the IH or management station closes the connection, either normally or in error, the MIH terminates the forwarding of that session.

It should be noted that the management station may choose to open a single SSH session to the MIH and utilize the SSH channel concept to establish multiple NETCONF sessions under a single SSH session. The NETCONF sessions could be directed to the MIH as well as any IH. This is an optimization that saves memory for the rather large SSH session state on the management station. For more information on SSH channels, see section 5 of RFC 4254.

The MIH will however have to establish full SSH sessions to each IH as forward requests come in from the management station.

25.4.3. Example - ConfD is configured to do automatic login

This is the most simple example, the manager sends a "forward" rpc and receives the capabilities of the IH.

Example 25.12. Auto login

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>board-1</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <data>
```

```
    <capabilities>
      <capability>urn:ietf:params:netconf:base:1.0</capability>
    </capabilities>
  </data>
</rpc-reply>
```

25.4.4. Example - Client needs to authenticate to the IH

Here the client sends a "forward" rpc and receives an error:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="1">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>board-1</target>
  </forward>
</rpc>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="1">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-app-tag>sasl-mechanisms</error-app-tag>
    <error-info>
      <mechanisms xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <mechanism>PLAIN</mechanism>
      </mechanisms>
    </error-info>
  </rpc-error>
</rpc>
```

The error indicates that the client needs to authenticate. This is done using the SASL protocol.

Example 25.13. Forward rpc with auth data

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="2">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>board-1</target>
    <auth>
      <mechanism>PLAIN</mechanism>
      <initial-response>AGFkbWluAHNlY3JldA==</initial-response>
    </auth>
  </forward>
</rpc>
```

The decoded initial response in the auth message is:

```
[NUL]admin[NUL]secret
```

Finally the client receives the capabilities of the IH

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="2">
  <data>
    <capabilities>
      <capability>urn:ietf:params:netconf:base:1.0</capability>
    </capabilities>
```

```
  </data>
</rpc-reply>
```

The client is now successfully connected to board-1

25.4.5. Example - Client needs to authenticate to the IH but fails

Similar to the example above, but the client sends a a bad password as in:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     message-id="2">
  <forward xmlns="http://tail-f.com/ns/netconf/forward/1.0">
    <target>board-1</target>
    <auth>
      <mechanism>PLAIN</mechanism>
      <initial-response>AGFkbWluAGFlY3JldA==</initial-response>
    </auth>
  </forward>
</rpc>
```

The decoded initial response in the auth message is:

```
[NUL]admin[NUL]aecret
```

An error is received:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           message-id="2">
  <rpc-error>
    <error-type>protocol</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-app-tag>sasl-failure</error-app-tag>
    <error-info>
      <failure xmlns="http://tail-f.com/ns/netconf/forward/1.0">
        <not-authorized/>
      </failure>
    </error-info>
  </rpc-error>
</rpc-reply>
```

Chapter 26. Plug-and-play scripting

26.1. Introduction

This chapter defines a scripting mechanism to be used together with the CLI (scripting is not available for any other northbound interfaces). The chapter is intended for users that are familiar with UNIX shell scripting and/or programming. With the scripting mechanism it is possible for an end-user to add new functionality to ConfD in a plug-and-play like manner. No special tools are needed. There are three categories of scripts:

`command scripts` — used to add new commands to the CLI.

`policy scripts` — invoked at validation time and may control the outcome of a transaction. Policy scripts have the mandate to cause a transaction to abort.

`post-commit scripts` — invoked when a transaction has been committed. Post-commit scripts can for example be used for logging, sending external events etc.

The terms "script" and "scripting" used throughout this description refer to how functionality can be added without a requirement for integration using the ConfD programming APIs. ConfD will only run the scripts as UNIX executables. Thus they may be written as shell scripts, or using some other scripting language that is supported by the OS, or even be compiled code. The scripts are run with the same user id as ConfD.

26.2. Script storage

Scripts are stored in a directory tree with a predefined structure where there is a sub-directory for each script category:

```
scripts/
        command/
        policy/
        post-commit/
```

For all script categories it suffices to just add a valid script in the correct sub-directory in order to enable the script. See the details for each script category for how a valid script of that category is defined. Scripts with a name beginning with a dot character ('.') are ignored.

The directory path to the location of the scripts is configured with the */confdConf/scripts/dir* configuration parameter. It is possible to have several scripts directories.

26.3. Script interface

All scripts are required to provide a formal description of their interface. When the scripts are loaded, ConfD will invoke the scripts with (one of)

`--command`

`--policy`

`--post-commit`

as argument depending of the script category.

The script must respond by writing its formal interface description on `stdout` and exit normally. Such a description consists of one or more sections. Which sections that are required depends on the category of the script.

The sections do however have a common syntax. Each section begins with the keyword "begin" followed by the type of section. After that one or more lines of settings follows. Each such setting begins with a name, followed by a colon character (':') and after that the value is stated. The section ends with the keyword "end". Empty lines and spaces may be used to improve the readability.

For examples see each corresponding section below.

26.4. Loading of scripts

Scripts are automatically loaded at startup and may also be manually reloaded with the CLI command **script reload**. The command takes an optional *`verbosity`* parameter which may have one of the following values:

`diff` Shows info about those scripts that have been changed since the latest (re)load. This is the default.

`all` Shows info about all scripts regardless of whether they have been changed or not.

`errors` Shows info about those scripts that are erroneous, regardless of whether they have been changed or not. Typical errors are invalid file permissions and syntax errors in the interface description.

Yet another parameter may be useful when debugging reload of scripts:

`debug` Shows additional debug info about the scripts.

An example session reloading scripts:

```
admin@ncs# script reload all
$NCS_DIR/examples.ncs/getting-started/using-ncs/7-scripting/scripts:
ok
command:
    add_user.sh: unchanged
    echo.sh: unchanged
policy:
    check_dir.sh: unchanged
post-commit:
    show_diff.sh: unchanged
/opt/ncs/scripts: ok
command:
    device_brief.sh: unchanged
    device_brief_c.sh: unchanged
    device_list.sh: unchanged
    device_list_c.sh: unchanged
    device_save.sh: unchanged
```

26.5. Command scripts

Command scripts are used to add new commands to the CLI. The scripts are executed in the context of a transaction. When the script is run in `oper` mode, this is a read-only transaction, when it is run in `config`

mode, it is a read-write transaction. In that context the script may make use of the environment variables `CONFD_MAAPI_USID` and `CONFD_MAAPI_THANDLE` in order to attach to the active transaction. This makes it simple to make use of the **maapi** command (see the ???? manual page) for various purposes.

Each command script must be able to handle the argument `--command` and, when invoked, write a `command` section to `stdout`. If the CLI command is intended to take parameters, one `param` section per CLI parameter must also be emitted.

26.5.1. Command section

The following settings can be used to define a command:

`modes` Defines in which CLI mode(s) that the command should be available. The value can be `oper`, `config` or both (separated with space).

`styles` Defines in which CLI styles that the command should be available. The value can be one or more of `c`, `i` and `j` (separated with space). `c` means Cisco style, `i`, means Cisco IOS and `j` for J-style.

`cmdpath` Is the full CLI command path. For example the command path `my script echo` implies that the command will be called `my script echo` in the CLI.

`help` Command help text.

An example of a command section is:

```
begin command
  modes: oper
  styles: c i j
  cmdpath: my script echo
  help: Display a line of text
end
```

26.5.2. Param section

In this section various aspects of a parameter is specified. This may both affect the parameter syntax for the end-user in the CLI as well as what the command script will get as arguments. The following settings can be used to customize each CLI parameter:

`name` Optional name of the parameter. If provided, the CLI will prompt for this name before the value. By default the name is not forwarded to the script. See `flag` and `prefix`.

`type` The type of the parameter. By default each parameter has a value, but by setting the type to `void` the CLI will not prompt for a value. In order to be useful the `void` type must be combined with `name` and either `flag` or `prefix`.

`presence` Controls whether the parameter must be present in the CLI input or not. Can be set to `optional` or `mandatory`.

`words` Controls the number of words that the parameter value may consist of. By default the value must consist of just one word (possibly quoted if it contains spaces). If set to `any`, the parameter may consist of any number of words. This setting is only valid for the last parameter.

`flag` Extra word added before the parameter value. For example if set to `-f` and the user enters `logfile`, the script will get `-f logfile` as arguments.

`prefix` Extra string prepended to the parameter value (as a single word). For example if set to `--file=` and the user enters `logfile`, the script will get `--file=logfile` as argument.

`help` Parameter help text.

If the command takes a parameter to redirect the output to a file, a param section might look like this:

```
begin param
 name: file
 presence: optional
 flag: -f
 help: Redirect output to file
end
```

26.5.3. Full command example

A command denying changes the configured trace-dir for a set of devices it can use the `check_dir.sh` script.

```
#!/bin/bash

set -e

while [ $# -gt 0 ]; do
    case "$1" in
        --command)
            # Configuration of the command
            #
            # modes   - CLI mode (oper config)
            # styles  - CLI style (c i j)
            # cmdpath - Full CLI command path
            # help    - Command help text
            #
            # Configuration of each parameter
            #
            # name     - (optional) name of the parameter
            # more     - (optional) true or false
            # presence - optional or mandatory
            # type     - void - A parameter without a value
            # words    - any - Multi word param. Only valid for the last param
            # flag     - Extra word added before the parameter value
            # prefix   - Extra string prepended to the parameter value
            # help     - Command help text
            cat << EOF

begin command
  modes: config
  styles: c i j
  cmdpath: user-wizard
  help: Add a new user
end
EOF
            exit
            ;;
        *)
            break
```

```
            ;;
    esac
    shift
done

## Ask for user name
while true; do
    echo -n "Enter user name: "
    read user

    if [ ! -n "${user}" ]; then
        echo "You failed to supply a user name."
    elif ncs-maapi --exists "/aaa:aaa/authentication/users/user{${user}}"; then
        echo "The user already exists."
    else
        break
    fi
done

## Ask for password
while true; do
    echo -n "Enter password: "
    read -s pass1
    echo

    if [ "${pass1:0:1}" == "$" ]; then
        echo -n "The password must not start with $. Please choose a "
        echo    "different password."
    else
        echo -n "Confirm password: "
        read -s pass2
        echo

        if [ "${pass1}" != "${pass2}" ]; then
            echo "Passwords do not match."
        else
            break
        fi
    fi
done

groups=`ncs-maapi --keys "/nacm/groups/group"`
while true; do
    echo "Choose a group for the user."
    echo -n "Available groups are: "
    for i in ${groups}; do echo -n "${i} "; done
    echo
    echo -n "Enter group for user: "
    read group

    if [ ! -n "${group}" ]; then
        echo "You must enter a valid group."
    else
        for i in ${groups}; do
            if [ "${i}" == "${group}" ]; then
                # valid group found
                break 2;
            fi
        done
        echo "You entered an invalid group."
```

```
    fi
    echo
done

echo "Creating user"

ncs-maapi --create "/aaa:aaa/authentication/users/user{${user}}"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/password" \
               "${pass1}"

echo "Setting home directory to: /homes/${user}"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/homedir" \
            "/homes/${user}"

echo "Setting ssh key directory to: /homes/${user}/ssh_keydir"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/ssh_keydir" \
            "/homes/${user}/ssh_keydir"

ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/uid" "1000"
ncs-maapi --set "/aaa:aaa/authentication/users/user{${user}}/gid" "100"

echo "Adding user to the ${group} group."
gusers=`ncs-maapi --get "/nacm/groups/group{${group}}/user-name"`

for i in ${gusers}; do
    if [ "${i}" == "${user}" ]; then
        echo "User already in group"
        exit 0
    fi
done

ncs-maapi --set "/nacm/groups/group{${group}}/user-name" "${gusers} ${user}"
```

Calling `examples.confd/scripting/scripts/command/echo.sh` with the argument `--command` argument produces a command section and a couple of param sections:

```
$ ./echo.sh --command
begin command
  modes: oper
  styles: c i j
  cmdpath: my script echo
  help: Display a line of text
end

begin param
 name: nolf
 type: void
 presence: optional
 flag: -n
 help: Do not output the trailing newline
end

begin param
 name: file
 presence: optional
 flag: -f
```

```
 help: Redirect output to file
end

begin param
 presence: mandatory
 words: any
 help: String to be displayed
end
```

In the complete example `examples.confd/scripting` there is a `README` file and a simple command script `scripts/command/echo.sh`.

26.6. Policy scripts

Policy scripts are invoked at validation time, before a change is committed. They provide a simplified way of defining validation points with callbacks. A policy script can reject the data, accept it, or accept it with a warning. If a warning is produced, it will be displayed for interactive users (e.g. through the CLI or Web UI). The user may choose to abort or continue to commit the transaction.

Policy scripts are typically assigned to individual leafs or containers. In some cases it may be feasible to use a single policy script, e.g. on the top level node of the configuration. In such a case, this script is responsible for the validation of all values and their relationships throughout the configuration.

All policy scripts are invoked on every configuration change. The policy scripts can be configured to depend on certain subtrees of the configuration, which can save time but it is very important that all dependencies are stated and also updated when the validation logic of the policy script is updated. Otherwise an update may be accepted even though a dependency should have denied it.

There can be multiple dependency declarations for a policy script. Each declaration consists of a dependency element specifying a configuration subtree that the validation code is dependent upon. If any element in any of the subtrees is modified, the policy script is invoked. A subtree is specified as an absolute path.

If there are no declared dependencies, the root of the configuration tree (/) is used, which means that the validation code is executed when any configuration element is modified. If dependencies are declared on a leaf element, an implicit dependency on the leaf itself is added.

Each policy script must handle the argument `--policy` and, when invoked, write a `policy` section to `stdout`. The script must also perform the actual validation when invoked with the argument `--keypath`.

26.6.1. Policy section

The following settings can be used to configure a policy script:

`keypath` Mandatory. Keypath is a path to a node in the configuration data tree. The policy script (and the automatically created validation point) will be associated with this node. The path must be absolute. A keypath can for example be `/devices/device/c0`. The script will be invoked if the configuration node, referred to by by the keypath, is changed or if any node in the subtree under the node (if the node is a container or list) is changed.

`dependency` Declaration of a dependency. The dependency must be an absolute keypath. Multiple dependency settings can be declared. Default is `/`.

`priority` An optional integer parameter specifying the order policy scripts and other validation callbacks will be evaluated, in order of increasing priority, where lower value is higher priority. The default priority is `0`.

`call` This optional setting can only be used if the associated node, declared as `keypath`, is a list. If set to `once`, the policy script is only called once even though there exists many list entries in the data store. This is useful if we have a huge amount of instances or if values assigned to each instance have to be validated in comparison with its siblings. Default is `each`.

A policy that will be run for every change on or under /devices/device.

```
begin policy
 keypath: /devices/device
 dependency: /devices/global-settings
 priority: 4
 call: each
end
```

26.6.2. Validation

When ConfD has come to the conclusion that the policy script should be invoked to perform its validation logic, the script is invoked with the option `--keypath`. If the registered node is a leaf, its value will be given with the `--value` option. For example `--keypath /devices/device/c0` or if the node is a leaf `--keypath /devices/device/c0/address --value 127.0.0.1`.

Once the script has performed its validation logic it must exit with a proper status. The following exit statuses are valid:

`0` Validation ok. Vote for commit.

`1` When the outcome of the validation is dubious it is possible for the script to issue a warning message. The message is extracted from the script output on stdout. An interactive user has the possibility to choose to abort or continue to commit the transaction. Non-interactive users automatically vote for commit.

`2` When the validation fails it is possible for the script to issue an error message. The message is extracted from the script output on stdout. The transaction will be aborted.

26.6.3. Full policy example

A policy denying changes the configured trace-dir for a set of devices it can use the `check_dir.sh` script.

```
#!/bin/sh

usage_and_exit() {
    cat << EOF
Usage: $0 -h
       $0 --policy
       $0 --keypath <keypath> [--value <value>]

  -h                    display this help and exit
  --policy              display policy configuration and exit
  --keypath <keypath>   path to node
  --value <value>       value of leaf

Return codes:
```

```
  0 - ok
  1 - warning message is printed on stdout
  2 - error message   is printed on stdout
EOF
    exit 1
}

while [ $# -gt 0 ]; do
    case "$1" in
        -h)
            usage_and_exit
            ;;
        --policy)
            cat << EOF
begin policy
  keypath: /devices/global-settings/trace-dir
  dependency: /devices/global-settings
  priority: 2
  call: each
end
EOF
            exit 0
            ;;
        --keypath)
            if [ $# -lt 2 ]; then
                echo "<ERROR> --keypath <keypath> - path omitted"
                usage_and_exit
            else
                keypath=$2
                shift
            fi
            ;;
        --value)
            if [ $# -lt 2 ]; then
                echo "<ERROR> --value <value> - leaf value omitted"
                usage_and_exit
            else
                value=$2
                shift
            fi
            ;;
        *)
            usage_and_exit
            ;;
    esac
    shift
done

if [ -z "${keypath}" ]; then
    echo "<ERROR> --keypath <keypath> is mandatory"
    usage_and_exit
fi

if [ -z "${value}" ]; then
    echo "<ERROR> --value <value> is mandatory"
    usage_and_exit
fi

orig="./logs"
```

```
dir=${value}
# dir=`ncs-maapi --get /devices/global-settings/trace-dir`
if [ "${dir}" != "${orig}" ] ; then
    echo "/devices/global-settings/trace-dir: must retain it original value (${orig})"
    exit 2
fi
```

Trying to change that parameter would result in an aborted transaction

```
admin@ncs(config)# devices global-settings trace-dir ./testing
admin@ncs(config)# commit
Aborted: /devices/global-settings/trace-dir: must retain it original
value (./logs)
```

In the complete example `examples.confd/scripting` there is a `README` file and a simple policy script `scripts/policy/check_number_of_hosts.sh` .

26.7. Post-commit scripts

Post-commit scripts are run when a transaction has been committed, but before any locks have been released. The transaction hangs until the script has returned. The script cannot change the outcome of the transaction. Post-commit scripts can for example be used for logging, sending external events etc. The scripts run as the same user id as ConfD .

The script is invoked with `--post-commit` at script (re)load. In future releases it is possible that the `post-commit` section will be used for control of post-commit scripts behavior.

At post-commit, the script is invoked without parameters. In that context the script may make use of the environment variables `CONFD_MAAPI_USID` and `CONFD_MAAPI_THANDLE` in order to attach to the active (read-only) transaction.

This makes it simple to make use of the **maapi** command. Especially the command **maapi --keypath-diff /** may turn out to be useful, as it provides a listing of all updates within the transaction on a format that is easy to parse.

26.7.1. Post-commit section

All post-commit scripts must be able to handle the argument `--post-commit` and, when invoked, write an empty `post-commit` section to `stdout`:

```
begin post-commit
end
```

26.7.2. Full post-commit example

Assume the administrator of a system would want to have a mail each time a change is performed on the system, a script such as `mail_admin.sh`:

```
#!/bin/bash
```

```
set -e

if [ $# -gt 0 ]; then
    case "$1" in
        --post-commit)
            cat &lt;&lt; EOF
begin post-commit
end
EOF
            exit 0
            ;;
        *)
            echo
            echo "Usage: $0 [--post-commit]"
            echo
            echo "  --post-commit Mandatory for post-commit scripts"
            exit 1
            ;;
    esac
else
    file="mail_admin.log"
    NCS_DIFF=$(ncs-maapi --keypath-diff /)
    mail -s "NCS Mailer" admin@example.com &lt;&lt;EOF
AutoGenerated mail from NCS

$NCS_DIFF
EOF
fi
```

If the admin then loads this script

```
admin@ncs# script reload debug
$NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/scripts:
ok
    post-commit:
        mail_admin.sh: new
--- Output from
$NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios/scripts/post-commit/m
--post-commit ---
1: begin post-commit
2: end
3:
---
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# devices global-settings trace-dir ./again
admin@ncs(config)# commit
Commit complete.
```

This configuration change will produce an email to admin@example.com with subject `NCS Mailer` and body

```
AutoGenerated mail from NCS
value set  : /devices/global-settings/trace-dir
```

In the complete example `examples.confd/scripting` there is a `README` file and a simple post-commit script `scripts/post-commit/show_diff.sh`.

Chapter 27. Advanced Topics

27.1. Datastores

A datastore is a complete set of configuration parameters for the device stored and manipulated as a single entity.

A datastore can be locked in its entirety with a global write lock.

ConfD supports three different named configuration datastores - *running*, *startup*, and *candidate*. The respective datastores support a set of capabilities as explained below:

running
: The *running* datastore contains the complete configuration currently active on the device. Running can be configured to support the *read-write* or the *writable-through-candidate* modes. Writable-through-candidate means that running can only be modified by making changes to the candidate datastore (see below), and by committing these changes to the candidate.

startup
: The *startup* datastore is a persistent datastore which the device reads every time it reboots.

  If running is read-write and the device has a startup datastore, a manager can try changes by writing them to running. If things look good, the changes can be made persistent by copying them to startup. This ensures that the device uses the same configuration after reboot.

candidate
: The *candidate* datastore is used to hold configuration data that can be manipulated without impacting the current configuration. The candidate configuration is a full configuration data set that serves as a workspace for creating and manipulating configuration data. Additions, deletions, and changes may be made to this data to construct the desired configuration.

  The candidate datastore can be committed, which means that the device's running configuration is replaced with the contents of the candidate datastore.

  The candidate can be used in two different modes, with different characteristics:

  - It can be modified without first taking a lock on the datastore. If it is modified outside a lock, it is marked as being *dirty*. When the candidate is dirty it means that it is (potentially) different from the running configuration. When it is dirty, a lock cannot be taken. It leaves the dirty state by being committed to running, or by discarding all changes (which effectively resets it to the contents of running).

  - If the candidate is not dirty, and a lock is taken, no one but the owner of the lock can modify the database. If changes are made to the candidate while it is locked, and the owner unlocks it (or closes the CLI, Web UI or NETCONF session), all changes are discarded, and the datastore is unlocked.

  The candidate can be committed to running with a specified *timeout*. In this case, running is set to the contents of the candidate. If a second commit, called a *confirming-commit*, is given within the timeout, the changes are made permanent. If no confirming-commit is given within the timeout period, running is reverted to the state it had before the first commit.

A project using ConfD must choose a valid combination of datastores to support. Which combination to choose depends on the system resources available on the device, and which characteristics the end-product should have.

The following is a list of valid combinations:

running in read-write mode, no startup, no candidate

- A single, non-volatile datastore is used.
- Once changes are written to the datastore, they are persistent, and cannot automatically be rolled back.
- The application needs to react to changes to the database. If CDB is used, this means that the application must use the subscription mechanism.

running in read-write mode and startup

- startup is stored in non-volatile memory, and running in read-write RAM.
- The application needs to be written in such a way that it reacts to changes to the database. If CDB is used, this means that the application must use the subscription mechanism.

running in read-write mode and candidate

- Both running and candidate are stored in non-volatile memory.
- NOTE: This combination is NOT RECOMMENDED. When a manager reconfigures a node that has the candidate and also read-write running, the manager can never know that running is up to date with the candidate and must thus always (logically) copy running to the candidate prior to modifying the candidate. This introduces unnecessary overhead, and makes automation more complicated.
- The application needs to react to changes to the database. If CDB is used, this means that the application must use the subscription mechanism.
- In this mode, running can be modified without going through the candidate. This means that a client that wishes to work with the candidate may need to copy running into the candidate, to ensure that no changes to running are lost when the candidate is committed.

running in writable-through-candidate mode and candidate

- Both running and candidate are stored in non-volatile memory, but the candidate can efficiently be implemented as a diff against running.
- The application needs to react to changes to the database. If CDB is used, this means that the application must use the subscription mechanism.
- In this mode, all changes always go through the candidate, so a client does never have to copy running to candidate in order to not lose any data.

ConfD ensures that running and startup are always consistent, in the sense that the validation constraints defined in the data model hold. The candidate is allowed to be temporarily inconsistent, but if it is committed to running, it must be valid.

ConfD by default implements the datastores chosen in CDB. However, ConfD can also be configured to use an external database. If an external database is used, this database *must* implement the running and startup datastores if applicable. If the candidate is used, it may be implemented with CDB or as an external database.

27.2. Locks

This section will explain the different locks that exist in ConfD and how they interact. It is important to understand the architecture of ConfD with its management backplane, and the transaction state machine as described in Section 7.5, “User sessions and ConfD Transactions” to be able to understand how the different locks fit into the picture.

27.2.1. Global locks

The ConfD management backplane keeps a lock for each datastore: running, startup and candidate. These locks are usually referred to as the global locks and they provide a mechanism to grant exclusive access to the datastore the lock guards.

The global locks are the only locks that can explicitly be taken through a northbound agent, for example by the NETCONF `<lock>` operation, or by calling `maapi_lock()`.

A global lock can be taken for the whole datastore, or it can be a partial lock (for a subset of the datamodel). Partial locks are exposed through NETCONF and MAAPI.

An agent can request a global lock to ensure that it has exclusive write-access to a datastore. When a global lock is held by an agent it is not possible for anyone else to write to the datastore the lock guards - this is enforced by the transaction engine. A global lock on a datastore is granted to an agent if there are no other holders of it (including partial locks), and if all dataproviders approve the lock request. Each dataprovider (CDB and/or external dataproviders) will have its `lock()` callback invoked to get a chance to refuse or accept the lock. The output of **confd --status** includes locking status. For each user session locks (if any) per datastore is listed.

27.2.2. Transaction locks

A northbound agent starts a user session towards ConfD's management backplane. Each user session can then start multiple transactions. A transaction is either read/write or read-only and is always started against a specific datastore.

The transaction engine has its internal locks, one for every datastore. These transaction locks exists to serialize configuration updates towards the datastore and are separate from the global locks.

As a northbound agent wants to update a datastore with a new configuration it will implicitly grab and release the transactional lock corresponding to the datastore it is trying to modify. The transaction engine takes care of managing the locks, as it moves through the transaction state machine and there is no API that exposes the transactional locks to the northbound agents.

When the transaction engine wants to take a lock for a transaction (for example when entering the validate state) it first checks that no other transaction has the lock. Then it checks that no user session has a global lock on that datastore. Finally each dataprovider is invoked by its `trans_lock()` callback.

27.2.3. Northbound agents and global locks

In contrast to the implicit transactional locks, some northbound agents expose explicit access to the global locks. This is done a bit differently by each agent.

The management API exposes the global locks by providing `maapi_lock()` and `maapi_unlock()` functions (and the corresponding `maapi_lock_partial()` `maapi_unlock_partial()` for partial locking). Once a user session is established (or attached to) these functions can be called.

In the CLI the global locks are taken when entering different configure modes as follows:

configure exclusive	When the candidate datastore is enabled both the running and candidate global locks will be taken.
configure exclusive	When the candidate datastore is disabled and the startup datastore is enabled both running (if enabled) and startup global locks are taken.
configure private | shared	Does not grab any locks

The global locks are then kept by the CLI until either the configure mode is exited, or in the case of **commit confirmed <timeout>** the lock is released when it returns.

The Web UI behaves in the same way as the CLI (it presents three edit tabs called "Edit private", "Edit exclusive", and "Edit shared" which corresponds to the CLI modes described above).

The NETCONF agent translates the `<lock>` operation into a request for the global lock for the requested datastore. Partial locks are also exposed through the partial-lock rpc.

27.2.4. External data providers

Implementing the `lock()` and `unlock()` callbacks is not required of an external dataprovider. ConfD will never try to initiate the `trans_lock()` state transition (see the transaction state diagram in Section 7.5, “User sessions and ConfD Transactions”) towards a data provider while a global lock is taken - so the reason for a dataprovider to implement the locking callbacks is if someone else can write (or lock for example to take a backup) to the data providers database.

27.2.5. CDB

CDB ignores the `lock()` and `unlock()` callbacks (since the data-provider interface is the only write interface towards it).

CDB has its own internal locks on the database. The running and startup datastore each has a single write and multiple read locks. It is not possible to grab the write-lock on a datastore while there are active read-locks on it. The locks in CDB exists to make sure that a reader always gets a consistent view of the data (in particular it becomes very confusing if another user is able to delete configuration nodes in between calls to get_next() on YANG list entries).

During a transaction `trans_lock()` takes a CDB read-lock towards the transactions datastore and write_start() tries to release the read-lock and grab the write-lock instead.

A CDB external client (usually referred to as an MO, managed object) implicitly takes a CDB read-lock between `cdb_start_session()` and `cdb_end_session()` on the specified datastore (running or startup). This means that while an MO is reading, a transaction can not pass through `write_start()` (and conversely a CDB reader can not start while a transaction is in between `write_start()` and `commit()` or `abort()`).

The Operational store in CDB does not have any locks. ConfD's transaction engine can only read from it, and the MO writes are atomic per write operation.

27.2.6. Lock impact on user sessions

When a session tries to modify a data store that is locked in some way, it will fail. For example, the CLI might print:

```
admin@host% commit
Aborted: the configuration database is locked
[error][2009-06-11 16:27:21]
```

Since some of the locks are short lived (such as a CDB read lock), ConfD can be configured to retry the failing operation for a short period of time. If the data store still is locked after this time, the operation fails.

To configure this, set `/confdConfig/commitRetryTimeout` in `confd.conf`.

27.3. Installing ConfD on a target system

The ConfD installation package contains both binaries for the target system and a development environment including documentation. Many of these files are not needed on a target, and can be excluded. Additional files can be removed depending on the feature configuration on the target.

In the following description, `$CONFD_DIR` refers to the directory where ConfD has been installed.

A minimal example set of files on a target system can be:

```
$CONFD_DIR/bin/confd
$CONFD_DIR/bin/confd_cli
$CONFD_DIR/etc/confd/*
$CONFD_DIR/lib/confd/bin/confd.boot
$CONFD_DIR/lib/confd/lib/core/*
$CONFD_DIR/lib/confd/lib/cli/*
```

`$CONFD_DIR/bin/confd_cli` is the command line interface (CLI) agent program and can be removed together with `$CONFD_DIR/lib/confd/lib/cli/` if the CLI is not used.

`$CONFD_DIR/etc/confd/` contains configuration files.

Support for Symmetric Multiprocessing (SMP) introduces some overhead both in CPU and memory usage, and in order to give optimal performance in all scenarios, the installation includes two separate executables for the ConfD daemon, `$CONFD_DIR/lib/confd/erts/bin/confd` (no SMP support) and `$CONFD_DIR/lib/confd/erts/bin/confd.smp` (with SMP support). If ConfD will always be run either with or without SMP support, one of these executables can be removed. See also the `--smp` option in the confd(1) manual page

If `$CONFD_DIR/lib/confd/erts/bin/confd` is removed, ConfD will always run with SMP support, although with a single thread on a single-processor system or if it is started with `--smp 1`. If `$CONFD_DIR/lib/confd/erts/bin/confd.smp` is removed, ConfD will never run with SMP support, and the `--smp` option has no effect other than refusing to start the daemon if the argument is bigger than 1.

Files associated with certain features can be removed if the system is set up not to use them:

The CLI agent	`$CONFD_DIR/bin/confd_cli` `$CONFD_DIR/lib/confd/lib/cli/`
The NETCONF server	`$CONFD_DIR/lib/confd/lib/netconf/`
The Web UI and REST server	`$CONFD_DIR/lib/confd/lib/webui/`
The Web UI frontend	`$CONFD_DIR/var/confd/webui/`

The SNMP agent and gateway	`$CONFD_DIR/bin/smidump`
	`$CONFD_DIR/lib/confd/lib/snmp/`
	
	smidump is only used for producing YANG files - it is not used by ConfD itself, and therefore not likely to be needed on the target.

The integrated SSH server	The integrated SSH server is not needed if OpenSSH is used to terminate SSH for NETCONF and the CLI: `$CONFD_DIR/lib/confd/lib/core/ssh*`

The confdc compiler	The compiler can be removed unless we plan to to compile YANG files on the host.
	
	`$CONFD_DIR/bin/confdc`
	`$CONFD_DIR/lib/confd/lib/confdc`

The AAA bridge	See documentation on AAA - basically this is a pre-compiled example program which probably won't be used on target: `$CONFD_DIR/lib/confd/lib/core/capi/priv/confd_aaa_bridge`

27.4. Configuring ConfD

When ConfD is started, it reads its configuration file and starts all subsystems configured to start (such as NETCONF, CLI etc.). If a configuration parameter is changed, ConfD can be reloaded by issuing:

```
$ confd --reload
```

This command also tells ConfD to close and reopen all log files, which makes it suitable to use from a system like **logrotate**.

There is also another way, whereby the ConfD configuration parameters that can be changed in runtime are loaded from an external namespace. Thus allowing the user to store ConfD's configuration in ConfD (specifically in CDB) itself. This will be described further down.

27.4.1. Using the configuration file

On a typical system, the configuration data resides in ConfD's database CDB. Some of the parameters in the configuration are intended for the target OS environment, such as the IP address of the management interface. The OS reads this information from its own configuration files, such as `/etc/conf.d/net`. This means that the application typically reads this data from CDB, and generates configuration files needed by the system before starting them. If a manager changes one of these parameters, the application subscribes to changes in CDB, regenerates the files, and restarts the system daemons. This mechanism can also be used for the configuration of ConfD itself. The application must subscribe to changes to any parameter affecting ConfD (such as management IP address), update the ConfD configuration file `confd.conf`, and then instruct ConfD to reload it.

ConfD comes bundled with a small example tool which can be used to patch `confd.conf` files: `$CONFD_DIR/src/confd/tools/xmlset.c`. This tool uses the light-weight Expat XML Parser (http://expat.sourceforge.net/).

This example changes `confd.conf` to disable the Web UI:

```
$ xmlset C false confdConfig webui enabled < confd.conf
```

This example changes `confd.conf` to removes the encryptedStrings container:

```
$ xmlset R confdConfig encryptedStrings < confd.conf
```

27.4.2. Storing ConfD configuration parameters in CDB

In the ConfD distribution in the `$CONFD_DIR/src/confd/dyncfg` directory the `confd_dyncfg.yang` YANG module is included. The module defines the namespace *http://tail-f.com/ns/confd_dyncfg/1.0* which contains all the ConfD configuration parameters that can be modified in run-time. I.e. it is a subset of the namespace that defines the ConfD configuration file (`confd.conf`).

To enable the feature of storing ConfD's configuration in CDB the setting `/confdConfig/runtimeReconfiguration` has to be set to *namespace* in the configuration file. This instructs ConfD to read all its "static" configuration from the configuration file, and then load the rest of the configuration from the confd_dyncfg namespace (which must be served by CDB). A requirement is that the confd_dyncfg.fxs is in ConfD's *loadPath*. It is also advisable to have a suitable `_init.xml` file in ConfD's CDB directory.

The best way to understand how to use this feature is the example `confdconf/dyncfg` in the bundled example collection.

In most cases the interesting use of this feature is to be able to expose a particular aspect of ConfD's configuration to the end-user and hide the rest. This can be achieved by combining the use of the **--export none** flag when compiling the `confd_dyncfg.yang` module with the use of the symlink feature (exactly how they work are explained in Section 10.7, “Hidden Data”). The `snmpa/6-dyncfg` example in the example collection shows how to expose a small subset of the SNMP agent configuration (as well as some minor aspects of the CLI parameters) in a private namespace.

For example, if we want to be able to expose the ConfD's built-in SNMP agents listen port as an end-user configurable as the leaf `/sys/snmp-port`, we could write a YANG model like this:

```
container sys {
        tailf:symlink snmp-port {
          tailf:path "/dyncfg:confdConfig/snmpAgent/port";
        }
      }
```

When a transaction containing changes to `/confdConfig` is committed ConfD will pick up the changes made and act accordingly. Thus there is no longer a need for **confd --reload** except for closing/re-opening of log-files (as described above) or to update the fxs files for sub-agents.

When `/confdConfig/runtimeReconfiguration` is set to *namespace*, any settings in `confd.conf` for the parameters that exist in the `confd_dyncfg` namespace are ignored, with one exception: the configuration under `/confdConfig/logs`. This configuration is needed before CDB has started, and ConfD will therefore initially use the settings from `confd.conf`, with the CDB settings taking precedence once CDB has started (i.e. when the transition to phase1 is completed).

27.5. Starting ConfD

By default, ConfD starts in the background without an associated terminal. If it is started as **confd --foreground**, it starts in the foreground attached to the current terminal. This feature can be used to start ConfD from a process manager. In order to properly stop ConfD in the foreground case, close ConfD's standard input, or use **confd --stop** as usual. When ConfD is started in the foreground, the commands **confd --wait-phase0** and **confd --wait-started** can be used to synchronize the startup sequence. See below for more details.

If startup or candidate with confirming-commit is used, the system might need to use a configuration which is different from the previous running when it reboots. An example of this is if startup is used, and a

manager writes a configuration into running which renders the device unstable, and it is rebooted. It might be that the management IP address used by the OS is not the one that should be used (if it was changed before reboot). We'd like to be able to change this address in the OS configuration files before bringing up the interface. But we don't know the address until ConfD has been started, and ConfD itself needs to listen to this address. To solve this dilemma, ConfD's startup sequence can be split into several phases. The first phase brings up the ConfD daemon, but no subsystems that listen to the management IP address (such as NETCONF and CLI). This phase must be started after the loopback interface has been brought up, since the loopback interface is used to communicate between the application and ConfD.

It is also necessary to use the start phases when CDB is used and semantic validation via external callbacks has been implemented. CDB will validate the new configuration when ConfD is started without an existing database, as well as when a schema upgrade has caused configuration changes. This validation is done on the transition to phase1, which means that validation callbacks must be registered before this.

Note

If an application has both validation callbacks and other callbacks (e.g. data provider), and uses the same daemon structure and control socket through all the phases, it must register all the callbacks in phase0. This is because the `confd_register_done()` function (see confd_lib_dp(3)) must be called after all registrations are done, and no callbacks will be invoked before this function has been called. The tables below reflect this requirement, but it is also possible to register all callbacks in phase0, which may simplify the startup sequence (however CDB subscribers can not be added until phase1).

The sequence to start up the system should be like this:

1. bring up the loopback interface
2. **confd --start-phase0**
3. start applications that implement validation callbacks
4. **confd --start-phase1**
5. start remaining applications, read from CDB
6. potentially update `confd.conf` and do **confd --reload**
7. bring up the management interface
8. **confd --start-phase2**

Note that if ConfD is started without any parameters, it will bring up the entire system at once.

This table summarizes the different start-phases and what they do.

Table 27.1. ConfD Start Phases

Command line	When command returns ConfD has	After which *application* can/should
`confd --start-phase0`	• If upgrading or initializing CDB, created an initial transaction.	• If upgrading or initializing, the application can modify the initial transaction

Command line	When command returns ConfD has	After which *application* can/ should
		• Register validation callbacks • Possibly register external data-providers, transformations, etc (see Note above) • Setup notification sockets • Connect to HA
`confd --start-phase1`	• If upgrading or initializing CDB, committed initial transaction	
		• Make HA state transitions • Register remaining external data-providers, transformation callbacks, etc (see Note above) • Add CDB subscribers
`confd --start-phase2`	• Bound and started listening to NETCONF, CLI, Web UI, and SNMP addresses / ports • Allowed initiation of MAAPI user sessions	

This table summarizes the different start-phases when ConfD is started in the foreground.

Table 27.2. ConfD Start Phases, running in foreground

Command line	When command returns ConfD has	After which *application* can/ should
`confd --foreground --start-phase0`	This command never returns.	
`confd --wait-phase0`	• If upgrading or initializing CDB, created an initial transaction.	
		• If upgrading or initializing, the application can modify the initial transaction • Register validation callbacks • Possibly register external data-providers, transformations, etc (see Note above) • Setup notification sockets
`confd --start-phase1`	• If upgrading or initializing CDB, committed initial transaction	

Command line	When command returns ConfD has	After which *application* can/ should
		• Connect to HA • Register remaining external data-providers, transformation callbacks, etc (see Note above) • Add CDB subscribers
`confd --start-phase2`	• Bound and started listening to NETCONF, CLI, Web UI, and SNMP addresses / ports • Allowed initiation of MAAPI user sessions	

27.6. ConfD IPC

Client libraries connect to ConfD using TCP. We tell ConfD which address to use for these connections through the `/confdConfig/confdIpcAddress/ip` (default value 127.0.0.1) and `/confdConfig/confdIpcAddress/port` (default value 4565) elements in `confd.conf`. It is possible to change these values, but it requires a number of steps to also configure the clients. Also there are security implications, see section *Security issues* below.

Some clients read the environment variables `CONFD_IPC_ADDR` and `CONFD_IPC_PORT` to determine if something other than the default is to be used, others might need to be recompiled. This is a list of clients which communicate with ConfD, and what needs to be done when `confdIpcAddress` is changed.

Client	Changes required
Remote commands via the *confd* command	Remote commands, such as **confd --reload**, check the environment variables **CONFD_IPC_ADDR** and **CONFD_IPC_PORT**.
CDB and MAAPI clients	The address supplied to `cdb_connect()` and `maapi_connect()` must be changed.
Data provider API clients	The address supplied to `confd_connect()` must be changed.
confd_cli	The Command Line Interface (CLI) client, **confd_cli**, checks the environment variables `CONFD_IPC_ADDR` and `CONFD_IPC_PORT`. Alternatively the port can be provided on the command line (using the **-P** option). *NOTE:* **confd_cli** is provided as source, in *$CONFD_DIR/src/confd/cli*, so it is also possible to re-compile it using the new address as default.
Notification API clients	The new address must be supplied to `confd_notifications_connect()`

To run more than one instance of ConfD on the same host (which can be useful in development scenarios) each instance needs its own IPC port. For each instance set `/confdConfig/confdIpcAddress/port` in `confd.conf` to something different.

There are two more instances of ports that will have to be modified, NETCONF and CLI over SSH. The netconf (SSH and TCP) ports that ConfD listens to by default are 2022 and 2023 respectively. Modify `/confdConfig/netconf/transport/ssh` and `/confdConfig/netconf/transport/tcp`, either by disabling them or changing the ports they listen to. The CLI over SSH by default listens to 2024; modify `/confdConfig/cli/ssh` either by disabling or changing the default port.

27.6.1. Using a different IPC mechanism

We can set up ConfD to use a different IPC mechanism than TCP for the client library connections, as well as for the communication between ConfD nodes in a HA cluster. This can be useful e.g. in a chassis system where ConfD runs on a management blade, while the managed objects run on data processing blades that may not have a TCP/IP implementation.

There are several requirements that must be fulfilled by such an IPC mechanism:

- It must adhere to the standard socket API, with SOCK_STREAM semantics. I.e. it must provide an ordered, reliable byte stream, with connection management via the `connect()`, `bind()`, `listen()`, and `accept()` primitives.
- It must support non-blocking operations (requested via `fcntl(O_NONBLOCK)`), for `accept()` as well as for read and write operations. Ideally non-blocking `connect()` should also be supported, but this is not currently used by ConfD in this case.
- It must support the use of `poll()` for I/O multiplexing.

For ConfD to be able to use this mechanism without knowledge of address format etc, we must provide C code in the form of a shared object, which is dynamically loaded by ConfD. The interface between ConfD and the shared object code is defined in the `ipc_drv.h` file in the `$CONFD_DIR/src/confd/ipc_drv` directory in the release. The shared object must be named `ipc_drv_ops.so` and installed in the `$CONFD_DIR/lib/confd/lib/core/confd/priv` directory of the ConfD installation, see the sample Makefile in the `ipc_drv` directory. The interface is implemented via the `confd_ext_ipc_init()` function. This function must be provided by the shared object, and it must return a pointer to a callback structure defined in the shared object:

```
struct confd_ext_ipc_cbs {
    int (*getaddrinfo)(char *address,
                       int *family, int *type, int *protocol,
                       struct sockaddr **addr, socklen_t *addrlen,
                       char **errstr);
    int (*socket)(int family, int type, int protocol, char **errstr);
    int (*getpeeraddr)(int fd, char **address, char **errstr);  /* optional */
    int (*connect)(char *address, char **errstr);
    int (*bind)(char *address, char **errstr);
    void (*unbind)(int fd);                                     /* optional */
};
```

The structure must provide (i.e. have non-NULL function pointers for) either both of the `getaddrinfo()` and `socket()` callbacks, or both of the `connect()` and `bind()` callbacks - it may of course provide all of them. The `getpeeraddr()` and `unbind()` callbacks are optional. If both `getaddrinfo()` and `socket()` are provided, the shared object can also be used by applications using the C APIs to connect to ConfD (see e.g. the `confd_cmd.c` source code in the `$CONFD_DIR/src/confd/tools` directory).

All the callbacks except `unbind()` can report an error by returning `-1`, and in this case optionally provide an error message via the *`errstr`* parameter. If an error message is provided, *`errstr`* must point to dynamically allocated memory - ConfD will free it through a call to `free(3)` after reporting the error.

getaddrinfo() — This callback should parse the given text-format *address* (see below). If the parsing is successful, the callback should return 0 and provide data that can be used for the `socket()` callback and for the standard `bind(2)` and/or `connect(2)` system calls via the *family*, *type*, *protocol*, *addr*, and *addrlen* parameters. The structure pointed to by *addr* must be dynamically allocated - ConfD will free it after use through a call to `free(3)`.

socket() — This callback should create a socket, and if successful return the socket file descriptor.

getpeeraddr() — This optional callback should create a text representation of the address of the remote host/node connected via the socket *fd*, and if successful return 0 and provide the text-format address via the *address* parameter. The main purpose of the callback is to make it possible to use the `maapi_disconnect_remote()` function (see the confd_lib_maapi(3) manual page), but the provided address will also be used in e.g. HA status and notifications, and will be included in ConfD debug dumps.

connect() — This callback should create a socket, connect it to the given *address* (see below), and if successful return the socket file descriptor.

bind() — This callback should create a socket, bind it to the given *address* (see below), and if successful return the socket file descriptor.

unbind() — This is an optional callback that can be used if we need to do any special cleanup when a bound socket is closed. In this case the callback must also close the file descriptor - otherwise the function pointer can be set to NULL, and ConfD will close the file descriptor.

Two examples using this interface are provided in the `$CONFD_DIR/src/confd/ipc_drv` directory. One of them (`ipc_drv_unix.c`) uses AF_UNIX sockets, and implements only the `connect()`, `bind()`, and `unbind()` callbacks. The other (`ipc_drv_etcp.c`) actually uses standard AF_INET/AF_INET6 TCP sockets just like the "normal" ConfD IPC - this can be meaningful if we need to set some non-standard socket options such as Linux SO_VRF for all IPC sockets. This example implements the `getaddrinfo()`, `socket()`, and `getpeeraddr()` callbacks.

An older version of this interface (also defined in `ipc_drv.h`) used a `confd_ipc_init()` function and a `struct confd_ipc_cbs` callback structure. This interface is deprecated, but will continue to be supported. The main differences are that the old interface lacks the `getaddrinfo()`, `socket()`, and `getpeeraddr()` callbacks, and that any error message would be provided via a static `errstr` structure element.

To enable the use of this alternate IPC mechanism for the client library connections, we need to set `/confdConfig/confdExternalIpc/enabled` to "true" in `confd.conf`. This causes any settings for `/confdConfig/confdIpcAddress/ip` and `/confdConfig/confdIpcAddress/port` to be ignored, and we can instead specify the address to use in `/confdConfig/confdExternalIpc/address`. The address is given in text form, and ConfD passes it to the `getaddrinfo()`, `bind()`, and/or `connect()` callbacks without any interpretation.

If we want to use the alternate IPC for the inter-node HA communication, we can in the same way set `/confdConfig/ha/externalIpc/enabled` and `/confdConfig/ha/externalIpc/address` in `confd.conf`. Additionally the HA API uses a struct that holds a node address:

```
struct confd_ha_node {
    confd_value_t nodeid;
    int af;               /* AF_INET | AF_INET6 | AF_UNSPEC */
```

```
    union {              /* address of remote note */
        struct in_addr ip4;
        struct in6_addr ip6;
        char *str;
    } addr;
    char buf[128];       /* when confd_read_notification() and       */
                         /* confd_ha_status() populate these structs, */
                         /* if type of nodeid is C_BUF, the pointer   */
                         /* will be set to point into this buffer     */
    char addr_buf[128];  /* similar to the above, but for the address */
                         /* of remote node when using external IPC    */
                         /* (from getpeeraddr() callback for slaves)  */
};
```

When this struct is used to specify the address of the master in the `confd_ha_beslave()` call, the `af` element should be set to `AF_UNSPEC`, and the `str` element of the `addr` union should point to the text form of the master node's address. When the struct is used to deliver information from ConfD, in the HA event notifications and the result of a `confd_ha_status()` call, `af` will also be set to `AF_UNSPEC`, but `str` will be `NULL` for slave nodes unless a peer address has been provided via the `getpeeraddr()` callback.

The client changes we need to do are analogous to those listed in the table above for the case of using a different IP address and/or port for TCP - the differences are:

- Instead of `CONFD_IPC_ADDR` and `CONFD_IPC_PORT`, the environment variable `CONFD_IPC_EXTADDR` is used to specify the address. This should be in the same form as used in `confd.conf`, and if the variable is set it causes any `CONFD_IPC_ADDR` and `CONFD_IPC_PORT` settings to be ignored.

- The **confd_cli** program also needs to be told where to find the shared object that it should use for the `connect()` operation. This is done via the `CONFD_IPC_EXTSOPATH` environment variable, i.e. it typically needs to be set to `$CONFD_DIR/lib/confd/lib/core/confd/priv/ipc_drv_ops.so`.

- Provided that the `getaddrinfo()` and `socket()` callbacks are provided by the shared object, the **confd_cmd**, **confd_load**, and **maapi** commands included in the release can also use the shared object if the `CONFD_IPC_EXTSOPATH` environment variable is set. Otherwise these programs will assume that any setting of environment `CONFD_IPC_EXTADDR` is the pathname of an AF_UNIX socket.

As noted above, **confd_cli** is provided as source, so we can alternatively modify it to support the alternate IPC mechanism "natively". This is also the case for **confd_cmd**, **confd_load**, and **maapi**.

Note

If we rebuild **confd_cli** or the other commands from source, but want to *keep* the support for alternate IPC via the environment variables and shared object, the preprocessor macro `EXTERNAL_IPC` must be defined. This can be done by un-commenting the `#define` in the source, or by using a **-D** option to the compiler.

27.6.2. Restricting access to the IPC port

By default, the clients connecting to the ConfD IPC port are considered trusted, i.e. there is no authentication required, and we rely on the use of 127.0.0.1 for `/confdConfig/confdIpcAddress/ip` to prevent remote access. In case this is not sufficient, it is possible to restrict the access to the IPC port by configuring an access check.

The access check is enabled by setting the `confd.conf` element `/confdConfig/confdIpcAccessCheck/enabled` to "true", and specifying a filename for `/confdConfig/confdIpcAccessCheck/filename`. The file should contain a shared secret, i.e. a random character string. Clients connecting to the IPC port will then be required to prove that they have knowledge of the secret through a challenge handshake, before they are allowed access to the ConfD functions provided via the IPC port.

Note

Obviously the access permissions on this file must be restricted via OS file permissions, such that it can only be read by the ConfD daemon and client processes that are allowed to connect to the IPC port. E.g. if both the ConfD daemon and the clients run as root, the file can be owned by root and have only "read by owner" permission (i.e. mode 0400). Another possibility is to have a group that only the ConfD daemon and the clients belong to, set the group ID of the file to that group, and have only "read by group" permission (i.e. mode 040).

To provide the secret to the client libraries, and inform them that they need to use the access check handshake, we have to set the environment variable `CONFD_IPC_ACCESS_FILE` to the full pathname of the file containing the secret. This is sufficient for all the clients mentioned above, i.e. there is no need to change application code to support or enable this check.

Note

The access check must be either enabled or disabled for both the ConfD daemon and the clients. E.g. if `/confdConfig/confdIpcAccessCheck/enabled` in `confd.conf` is *not* set to "true", but clients are started with the environment variable `CONFD_IPC_ACCESS_FILE` pointing to a file with a secret, the client connections will fail.

27.7. Restart strategies

If the ConfD daemon is shut down, all applications connected to the ConfD daemon must enter an indefinite reconnect loop. If ConfD has been configured to use a startup datastore, all applications keeping configuration data in their run-time state *must* re-read the configuration data from CDB, when the daemon comes back.

If ConfD has been setup to *not* use a startup datastore, all applications which keep configuration data in their run-time state can just proceed its processing without any re-read of the configuration data from CDB, when the daemon comes back.

The ConfD daemon *must* be restarted if .fxs files in a running system are to be changed. It is not enough to issue a:

```
$ confd --reload
```

Before we restart the daemon we need to stop all applications relying on the .fxs files that are updated. Whenever the daemon is up and running the stopped applications can be restarted.

Applications which do not rely on the updated .fxs files can safely be kept running. However, be sure to follow the startup datastore reconnect strategy above.

27.8. Security issues

ConfD requires some privileges to perform certain tasks. The following tasks may, depending on the target system, require root privileges.

- Binding to privileged ports. The `confd.conf` configuration file specifies which port numbers ConfD should *bind(2)* to. If any of these port numbers are lower than 1024, ConfD usually requires root privileges unless the target operating system allows ConfD to bind to these ports as a non-root user.

- If PAM is to be used for authentication, the program installed as `$CONFD_DIR/lib/confd/lib/core/pam/priv/epam` acts as a PAM client. Depending on the local PAM configuration, this program may require root privileges. If PAM is configured to read the local `passwd` file, the program must either run as root, or be setuid root. If the local PAM configuration instructs ConfD to run for example *pam_radius_auth*, root privileges are possibly not required depending on the local PAM installation.

- If the CLI is used and we want to create CLI commands that run executables, we may want to modify the permissions of the `$CONFD_DIR/lib/confd/lib/core/confd/priv/cmdptywrapper` program.

  To be able to run an executable as root or a specific user, we need to make cmdptywrapper setuid root, i.e.:

  1. **# chown root cmdptywrapper**

  2. **# chmod u+s cmdptywrapper**

  Failing that, all programs will be executed as the user running the **confd** daemon. Consequently, if that user is root we do not have to perform the chmod operations above.

ConfD can be instructed to terminate NETCONF over clear text TCP. This is useful for debugging since the NETCONF traffic can then be easily captured and analyzed. It is also useful if we want to provide some local proprietary transport mechanism which is not SSH. Clear text TCP termination is not authenticated, the clear text client simply tells ConfD which user the session should run as. The idea is that authentication is already done by some external entity, such as an SSH server. If clear text TCP is enabled, it is very important that ConfD binds to localhost (127.0.0.1) for these connections.

Client libraries connect to ConfD. For example the CDB API is TCP based and a CDB client connects to ConfD. We instruct ConfD which address to use for these connections through the confd.conf parameters `/confdConfig/confdIpcAddress/ip` (default address 127.0.0.1) and `/confdConfig/confdIpcAddress/port` (default port 4565).

ConfD multiplexes different kinds of connections on the same socket (IP and port combination). The following programs connect on the socket:

- Remote commands, such as e.g. **confd --reload**

- CDB clients.

- External database API clients.

- MAAPI, The Management Agent API clients.

- The **confd_cli** program

All of the above are considered trusted. MAAPI clients and **confd_cli** should supposedly authenticate the user before connecting to ConfD whereas CDB clients and external database API clients are considered trusted and do not have to authenticate.

Thus, since the *confdIpcAddress* socket allows full unauthenticated access to the system, it is important to ensure that the socket is not accessible from untrusted networks. However it is also possible to restrict access to this socket by means of an access check, see Section 27.6.2, “Restricting access to the IPC port” above.

27.9. Running ConfD as a non privileged user

A common misfeature found on UN*X operating systems is the restriction that only root can bind to ports below 1024. Many a dollar has been wasted on workarounds and often the results are security holes.

Both FreeBSD and Solaris have elegant configuration options to turn this feature off. On FreeBSD:

```
$ sysctl net.inet.ip.portrange.reservedhigh=0
```

The above is best added to your /etc/sysctl.conf

Similarly on Solaris we can just configure this. Assuming we want to run ConfD under a non-root user "confd". On Solaris we can do that easily by granting the specific right to bind privileged ports below 1024 (and only that) to the "confd" user using:

```
$ /usr/sbin/usermod -K defaultpriv=basic,net_privaddr confd
```

And check the we get what we want through:

```
$ grep confd /etc/user_attr
confd::::type=normal;defaultpriv=basic,net_privaddr
```

Linux doesn't have anything like the above. There are a couple of options on Linux. The best is to use an auxiliary program like authbind `http://packages.debian.org/stable/authbind` or privbind `http://sourceforge.net/projects/privbind/`

These programs are run by root. To start confd under e.g authbind we can do:

```
privbind -u confd /opt/confd/confd-2.7/bin/confd \
    -c /etc/confd.conf
```

The above command starts confd as user *confd* and binds to ports below 1024

27.10. Storing encrypted values in ConfD

Using the `tailf:des3-cbc-encrypted-string` or the `tailf:aes-cfb-128-encrypted-string` built-in types it is possible to store encrypted values in ConfD (see confd_types(3)). The keys used to encrypt these values are stored in `confd.conf`. Whenever an encrypted leaf is read using the CDB API or MAAPI it is possible to decrypt the returned string using the `confd_decrypt()` function. When the keys in `confd.conf` are changed, the encrypted values will not be decryptable any longer, so care must be taken to re-install the values using the new keys. This section will provide an example on how to do this.

The encrypted values can only be decrypted using `confd_decrypt()`, which only works when ConfD is running with the correct keys, so the procedure to update the encrypted values is:

1. Read all the encrypted values and decrypt them

2. Stop the ConfD daemon

3. Restart it with the new encryption keys

4. Write back the values in clear-text, which will cause ConfD to encrypt them again

A very simple YANG model to store encrypted strings could be:

```
module enctest {
    namespace "http://www.example.com/ns/enctest";
    prefix e;
    import tailf-common {
        prefix tailf;
    }

    container strs {
        list str {
            key nr;
            max-elements 64;
            leaf nr {
                type int32;
            }
            leaf secret {
                type tailf:aes-cfb-128-encrypted-string;
                mandatory true;
            }
        }
    }
}
```

The we could write a function which would read all the encrypted leafs and save the clear-text equivalent. Such a function (without error checking) could look like this:

```
static void install_keys(struct sockaddr_in *addr)
{
    struct confd_daemon_ctx *dctx;
    int ctlsock = socket(PF_INET, SOCK_STREAM, 0);
    dctx = confd_init_daemon(progname);
    confd_connect(dctx, ctlsock, CONTROL_SOCKET, (struct sockaddr*)addr, sizeof (*addr));
    confd_install_crypto_keys(dctx);
    close(ctlsock);
    confd_release_daemon(dctx);
}

static void get_clear_text(struct sockaddr_in *addr, FILE *f)
{
    int rsock = socket(PF_INET, SOCK_STREAM, 0);
    int i, n;

    install_keys(addr);

    cdb_connect(rsock, CDB_READ_SOCKET, (struct sockaddr*)addr, sizeof(*addr));
    cdb_start_session(rsock, CDB_RUNNING);
    cdb_set_namespace(rsock, smp__ns);
    n = cdb_num_instances(rsock, "/strs/str");
    for(i=0; i<n; i++) {
        int nr;
        char cstr[BUFSIZ], dstr[BUFSIZ];
```

```
        cdb_get_str(rsock, cstr, sizeof(cstr), "/strs/str[%d]/secret", i);
        cdb_get_int32(rsock, &nr, "/strs/str[%d]/nr", i);
        memset(dstr, 0, sizeof(dstr));
        confd_decrypt(cstr, strlen(cstr), dstr);
        fprintf(f, "/strs/str{%d}/secret=$0$%s\n", nr, dstr);
    }
    cdb_end_session(rsock),
    cdb_close(rsock);
}
```

Note the prefixing of the clear-text output of `$0$` - this is what indicates to the ConfD daemon that the strings are in clear text, causing it to encrypt them when we install them again.

Now the opposite function, reading lines on the form `"keypath=value"` and using the `maapi_set_elem2()` function to write them back to the ConfD daemon.

```
static void set_values(struct sockaddr_in *addr, FILE *f)
{
    int msock = socket(PF_INET, SOCK_STREAM, 0);
    int th;
    struct confd_ip ip;
    const char *groups[] = { "admin" };

    maapi_connect(msock, (struct sockaddr*)addr, sizeof(*addr));
    ip.af = AF_INET;
    inet_aton("127.0.0.1", &ip.ip.v4);
    maapi_start_user_session(msock, "admin", progname,
                             groups, sizeof(groups) / sizeof(*groups),
                             &ip, CONFD_PROTO_TCP);

    maapi_start_trans(msock, CONFD_RUNNING, CONFD_READ_WRITE);
    maapi_set_namespace(msock, th, smp__ns);
    for (;;) {
        char *key, *val, line[BUFSIZ];
        if (fgets(line, sizeof(line), f) == NULL) {
            break;
        }
        key = line;
        val = strchr(key, (int)'=');
        *val++ = 0; /* NUL terminate the key, make val point to value */
        maapi_set_elem2(msock, th, val, key);
    }
    maapi_apply_trans(msock, th, 0);
    maapi_end_user_session(msock);
    close(msock);
}
```

Putting it together with this `main()` function makes a useful utility program for the task at hand.

```
int main(int argc, char **argv)
{
    char *confd_addr = "127.0.0.1";
    int confd_port = CONFD_PORT;
    struct sockaddr_in addr;
    int c, mode = 0;            /* 1 = get, 2 = set */

    /* Parse command line */
    while ((c = getopt(argc, argv, "gs")) != EOF) {
        switch (c) {
```

```
        case 'g':
            mode = 1;
            break;
        case 's':
            mode = 2;
            break;
        default:
            printf("huh?\n");
            exit(1);
        }
    }
    if (!mode) {
        fprintf(stderr, "%s: must provide either -s or -g\n", argv[0]);
        exit(1);
    }
    /* Initialize address to confd daemon */
    {
        struct in_addr in;
        inet_aton(confd_addr, &in);
        addr.sin_addr.s_addr = in.s_addr;
        addr.sin_family = AF_INET;
        addr.sin_port = htons(confd_port);
    }
    confd_init(argv[0], stderr, dbg);

    switch (mode) {
    case 1: get_clear_text(&addr, stdout); break;
    case 2: set_values(&addr, stdin); break;
    }
    exit(0);
}
```

Using this utility, called **crypto_keys**, installing new encryption keys could be done using a shell script like this.

```
# First save clear text version of the keys in a temporary file
crypto_keys -g > TOP_SECRET

# Now stop the daemon
confd --stop

# Install the new AES encryption key (provided to this script in $1)
mv confd.conf confd.conf.old
xmlset C "$1" confdConfig encryptedStrings AESCFB128 key < \
    confd.conf.old  > confd.conf
rm -f confd.conf.old

# Bring the daemon up to start-phase 1
confd -c confd-conf --start-phase0
confd --start-phase1

# Now write back the keys, and remove the temporary file
crypto_keys -s < TOP_SECRET
rm -f TOP_SECRET

# We are done
confd --start-phase2
```

In this example we are only using AES encryption, and only modifying the key, not the initial vector - but it is easy to extend to use the 3DES keys as well. The **xmlset** utility (provided as example

source in `$CONFD_DIR/src/confd/tools`) in the ConfD distribution) is used to modify the key in `confd.conf`. Writing back the encrypted leaf in start phase 1 ensures that no external method (e.g. a NETCONF request) modifies the data before it is re-installed with the new encryption keys.

27.11. Disaster management

This section describes a number of disaster scenarios and recommends various actions to take in the different disaster variants.

27.11.1. ConfD fails to start

CDB keeps its data in two files `A.cdb` and `C.cdb`. If ConfD is stopped, these two files can simply be copied, and the copy is then a full backup of CDB. If ConfD is running, we cannot copy the files, but need to use **confd --cdb-backup file** to copy the two CDB files into a backup file (in gzipped tar format).

Furthermore, if neither A.cdb nor C.cdb exists in the configured CDB directory, CDB will attempt to initialize from all files in the CDB directory with the suffix ".xml".

Thus, there exists two different ways to reinitiate CDB from a previous known good state, either from .xml files of from a CDB backup. The .xml files would typically be used to reinstall "factory defaults" whereas a CDB backup could be used in more complex scenarios.

When ConfD starts and fails to initialize, the following exit codes can occur:

- Exit codes *1* and *19* mean that an internal error has occurred. A text message should be in the logs, or if the error occurred at startup before logging had been activated, on standard error (standard output if ConfD was started with --foreground). Generally the message will only be meaningful to the ConfD developers, and an internal error should always be reported to Tail-f support.

- Exit codes *2* and *3* are only used for the confd "control commands" (see the section COMMUNICATING WITH CONFD in the confd(1) manual page), and mean that the command failed due to timeout. Code *2* is used when the initial connect to ConfD didn't succeed within 5 seconds (or the `TryTime` if given), while code *3* means that the ConfD daemon did not complete the command within the time given by the `--timeout` option.

- Exit code *10* means that one of the init files in the CDB directory was faulty in some way. Further information in the log.

- Exit code *11* means that the CDB configuration was changed in an unsupported way. This will only happen when an existing database is detected, which was created with another configuration than the current in confd.conf.

- Exit code *12* means that the C.cdb file is in an old and unsupported format (this can only happen if the CDB database was created with a ConfD version older than 1.3, from which upgrading isn't supported).

- Exit code *13* means that the schema change caused an upgrade, but for some reason the upgrade failed. Details are in the log. The way to recover from this situation is either to correct the problem or to re-install the old schema (fxs) files.

- Exit code *14* means that the schema change caused an upgrade, but for some reason the upgrade failed, corrupting the database in the process. This is rare and usually caused by a bug. To recover, either start from an empty database with the new schema, or re-install the old schema files and apply a backup.

- Exit code *15* means that `A.cdb` or `C.cdb` is corrupt in a non-recoverable way. Remove the files and re-start using a backup or init files.

- Exit code *16* means that CDB ran into an unrecoverable file-error while booting (such as running out of space on the device while writing the initial schema file).

- Exit code *20* means that ConfD failed to bind a socket. By default this means that Confd refuses to start. It is however possible to force Confd to ignore this fatal error situation by enabling the parameter */confdConfig/ignoreBindErrors*. Instead a warning is issued and the failing northbound agent is disabled. The agent may be enabled by dynamically re-configuring the failing agent to use another port and restart Confd.

- Exit code *21* means that some ConfD configuration file is faulty. More information in the logs.

- Exit code *22* indicates a ConfD installation related problem, e.g. that the user does not have read access to some library files, or that some file is missing.

If the ConfD daemon starts normally, the exit code is *0*.

If CDB is reinitialized to factory defaults, it may not be possible to reach the machine over the network. The only way to reconfigure the machine is through a CLI login over the serial console.

If the AAA database is broken, ConfD will start but with no authorization rules loaded. This means that all write access to the configuration is denied. The ConfD CLI can be started with a flag **confd_cli --noaaa** which will allow full unauthorized access to the configuration. Usage of the ConfD cli with this flag can possibly be enabled for some special UNIX user which can only login over the serial port. Thus **--noaaa** provides a way to reconfigure the box although the AAA database is broken.

27.11.2. ConfD failure after startup

ConfD attempts to handle all runtime problems without terminating, e.g. by restarting specific components. However there are some cases where this is not possible, described below. When ConfD is started the default way, i.e. as a daemon, the exit codes will of course not be available, but see the --foreground option in the *confd(1)* manual page.

- Out of memory: If ConfD is unable to allocate memory, it will exit by calling *abort(3)*. This will generate an exit code as for reception of the SIGABRT signal - e.g. if ConfD is started from a shell script, it will see 134 as exit code (128 + the signal number).

- Out of file descriptors for *accept(2)*: If ConfD fails to accept a TCP connection due to lack of file descriptors, it will log this and then exit with code 25. To avoid this problem, make sure that the process and system-wide file descriptor limits are set high enough, and if needed configure session limits in `confd.conf`.

27.11.3. Transaction commit failure

When the system is updated, ConfD executes a two phase commit protocol towards the different participating databases including CDB. If a participant fails in the `commit()` phase although the participant succeeded in the prepare phase, the configuration is possibly in an inconsistent state.

When ConfD considers the configuration to be in a inconsistent state, operations will continue. It is still possible to use NETCONF, the CLI and all other northbound management agents. The CLI has a different prompt which reflects that the system is considered to be in an inconsistent state and also the Web UI shows this:

```
-- WARNING -----------------------------------------------------
```

```
Running db may be inconsistent. Enter private configuration mode and
install a rollback configuration or load a saved configuration.
-----------------------------------------------------------------
```

It is slightly more involved using the NETCONF agent. The NETCONF transaction which resulted in a failed commit will fail, but following that the only way to see that the system is considered to be in an inconsistent state is by reading the data defined by *tailf-netconf-monitoring*.

The MAAPI API has two interface functions which can be used to set and retrieve the consistency status. This API can thus be used to manually reset the consistency state. Apart from this, the only way to reset the state to a consistent state is by reloading the entire configuration.

27.12. Troubleshooting

This section discusses problems that new users have seen when they started to use ConfD. Please do not hesitate to contact our support team (see below) if you are having trouble, regardless of whether your problem is listed here or not.

27.12.1. Installation Problems

Error messages during installation

The installation program gives a lot of error messages, the first few like the ones below. The resulting installation is obviously incomplete.

```
tar: Skipping to next header
gzip: stdin: invalid compressed data--format violated
```

Cause: This happens if the installation program has been damaged, most likely because it has been downloaded in 'ascii' mode.

Resolution: Remove the installation directory. Download a new copy of ConfD from our servers. Make sure you use binary transfer mode every step of the way.

27.12.2. Problems Starting ConfD

ConfD terminating with GLIBC error

ConfD terminates immediately with a message similar to the one below.

```
Internal error: Open failed: /lib/tls/libc.so.6: version
`GLIBC_2.3.4' not found (required by
.../lib/confd/lib/core/util/priv/syst_drv.so)
```

Cause: This happens if you are running on a very old Linux version. The GNU libc (GLIBC) version is older than 2.3.4, which was released 2004.

Resolution: Use a newer Linux system, or upgrade the GLIBC installation.

ConfD terminating with libcrypto error

- ConfD terminates immediately with a message similar to this:

```
Bad configuration: .../confd.conf:0: cannot dynamically link with
libcrypto shared library
```

Cause: This normally happens due to the OpenSSL package being of the wrong version or not installed in the operating system.

Resolution: One of

1. Install the OpenSSL package with the correct version. This is 1.0.0 for Linux releases of ConfD, 0.9.8 or 0.9.7 for some other operating systems. To find out the version to install, run:

   ```
   $ ldd $CONFD_DIR/lib/confd/lib/core/crypto/priv/lib/crypto.so
   ```

   Note: only the libcrypto shared library (libcrypto.so.N.N.N) is actually required by ConfD.

2. Provided that a different version of OpenSSL, 0.9.7 or greater, is installed: Rebuild the ConfD components that depend on libcrypto to use this version, as described in Section 27.15, “Using a different version of OpenSSL”.

- ConfD terminates immediately, or when the Web UI is enabled, with a message similar to:

  ```
  Bad configuration: .../confd.conf:0: libcrypto shared library mismatch
  (DES_INT) - crypto.so and libconfd must be rebuilt
  ```

  or:

  ```
  Bad configuration: .../confd.conf:0: libcrypto shared library mismatch
  (RC4_CHAR) - crypto.so must be rebuilt for support of default setting
  for /confdConfig/webui/transport/ssl/ciphers
  ```

  Cause: This happens if the OpenSSL package is of the correct version, but has been built with a configuration parameter that makes the interface incompatible with the build that is expected by ConfD.

  Resolution: Applying resolution 2 above is always sufficient. Applying resolution 1 is also a possibility, but requires that the OpenSSL package is built with the expected configuration parameters. Contact Tail-f support if this method is desired but unsuccessful in solving the problem. In case only the second message (with RC4_CHAR) occurs, yet another way to resolve the issue is to configure a cipher list for `/confdConfig/webui/transport/ssl/ciphers` in `confd.conf` (or confd_dyncfg) that does not include any RC4-based ciphers - see confd.conf(5).

27.12.3. Problems Running Examples

Some examples are dependent on features that might only be available on Linux. Before such examples can run, they would have to be ported.

The 'netconf-console' program fails

Sending NETCONF commands and queries with 'netconf-console' fails, while it works using 'netconf-console-tcp'. The error message is below.

You must install the python ssh implementation paramiko in order to use ssh.

Cause: The netconf-console command is implemented using the Python programming language. It depends on the python SSH implementation Paramiko. Since you are seeing this message, your operating system doesn't have the python-module Paramiko installed. The Paramiko package, in turn, depends on a Python crypto library (pycrypto).

Resolution: Install Paramiko (and pycrypto, if necessary) using the standard installation mechanisms for your OS. An alternative approach is to go to the project home pages to fetch, build and install the missing packages.

- http://www.lag.net/paramiko/
- http://www.amk.ca/python/code/crypto

These packages come with simple installation instructions. You will need root privileges to install these packages, however. When properly installed, you should be able to import the paramiko module without error messages

```
$ python
...
>>> import paramiko
>>>
```

Exit the Python interpreter with Ctrl+D.

A workaround is to use 'netconf-console-tcp'. It uses TCP instead of SSH and doesn't require Paramiko or Pycrypto. Note that TCP traffic is not encrypted.

27.12.4. General Troubleshooting Strategies

If you have trouble starting or running ConfD, the examples or the clients you write, here are some troubleshooting tips.

Transcript

When contacting support, it often helps the support engineer to understand what you are trying to achieve if you copy-paste the commands, responses and shell scripts that you used to trigger the problem.

Verbose flag

When ConfD is started, give the --verbose (abbreviated -v) and --foreground flags. This will prevent ConfD from starting as a daemon and cause some messages to be printed on the stdout.

```
$ confd --verbose --foreground ...
```

Log files

To find out what ConfD is/was doing, browsing ConfD's log files is often helpful. In the examples, they are called 'devel.log', 'confd.log', 'audit.log' and 'confd.log'. If you are working with your own system, make sure the log files are enabled in 'confd.conf'. They are already enabled in all the examples.

Status

ConfD will give you a comprehensive status report if you call

```
$ confd --status
```

ConfD status information is also available as operational data under `/confd-state` when the `tailf-confd-monitoring.fxs` and `tailf-common-monitoring.fxs` data model files are present in ConfD's `loadPath`. These files are stored in `$CONFD_DIR/etc/confd` in the ConfD release, and the functionality thus enabled by default. See the corresponding YANG modules `tailf-confd-`

`monitoring.yang` and `tailf-common-monitoring.yang` in the `$CONFD_DIR/src/confd/yang` directory of the ConfD release for documentation of the provided data. To allow programmatic access to this data via MAAPI without exposing it to end users, the modules can be recompiled with the `--export none` option to **confdc** (see confdc (1)).

> **Note**
>
> When recompiling these modules, it is critical that the annotation module `tailf-confd-monitoring-ann.yang` is used, see `$CONFD_DIR/src/confd/yang/Makefile`.

Check data provider

If you are implementing a data provider (for operational or configuration data), you can verify that it works for all possible data items using

```
$ confd --check-callbacks
```

Debug dump

If you suspect you have experienced a bug in ConfD, or ConfD told you so, you can give Support a debug dump to help us diagnose the problem. It contains a lot of status information (including a full confd --status report) and some internal state information. This information is only readable and comprehensible to the ConfD development team, so send the dump to your support contact. A debug dump is created using

```
$ confd --debug-dump mydump1
```

Just as in CSI on TV, it's important that the information is collected as soon as possible after the event. Many interesting traces will wash away with time, or stay undetected if there are lots of irrelevant facts in the dump.

Debug error log

Another thing you can do if you suspect you have experienced a bug in ConfD, is to enable the error log. The logged information is only readable and comprehensible to the ConfD development team, so send the log to your support contact.

By default, the error log is disabled. To enable it, add this chunk of XML between `<logs>` and `</logs>` in your `confd.conf` file:

```
<errorLog>
  <enabled>true</enabled>
  <filename>./error.log</filename>
</errorLog>
```

This will actually create a number of files called ./error.log*. Please send them all to us.

System dump

If ConfD aborts due to failure to allocate memory (see Section 27.11, “Disaster management”), and you believe that this is due to a memory leak in ConfD, creating one or more debug dumps as described above (before ConfD aborts) will produce the most useful information

for Support. If this is not possible, you can make ConfD produce a system dump just before aborting. To do this, set the environment variable `$CONFD_DUMP` to a file name for the dump before starting ConfD. The dumped information is only comprehensible to the ConfD development team, so send the dump to your support contact.

System call trace

To catch certain types of problems, especially relating to system start and configuration, the operating system's system call trace can be invaluable. This tool is called strace/ktrace/truss. Please send the result to your support contact for a diagnosis. Running instructions below.

Linux:

```
$ strace -f -o mylog1.strace -s 1024 confd ...
```

BSD:

```
$ ktrace -ad -f mylog1.ktrace confd ...
$ kdump -f mylog1.ktrace > mylog1.kdump
```

Solaris:

```
$ truss -f -o mylog1.truss confd ...
```

Application debugging

The primary tool for debugging the interaction between applications and ConfD is to give the debug level *`debug`* to `confd_init()` as `CONFD_TRACE`, see the confd_lib_lib(3) manual page. If more in-depth debugging using e.g. **gdb** is needed, it may be useful to rebuild the `libconfd` library from source with debugging symbols. This can be done by using the libconfd source package `confd-<vsn>.libconfd.tar.gz` that is delivered with the ConfD release. The package includes a `README` file that describes how to do the build - note in particular the "Application debugging" section.

When debugging application memory leaks with a tool like **valgrind**, it is often *necessary* to rebuild libconfd from source, since the default build uses a "pool allocator" that makes the stack trace information for memory leaks from **valgrind** completely misleading for allocations from libconfd. The details of how to do a build that disables the pool allocator are described in the "Application debugging" section of the `README` in the libconfd source package.

27.13. Tuning the size of confd_hkeypath_t

The ConfD C API library `libconfd` uses a C struct for passing keypaths to callback functions:

```
typedef struct confd_hkeypath {
    int len;
    confd_value_t v[MAXDEPTH][MAXKEYLEN];
} confd_hkeypath_t;
```

See the section called “XML PATHS” in the confd_types(3) manual page for discussion about how this struct is used. The values used for `MAXDEPTH` and `MAXKEYLEN` are 20 and 9, respectively, which should be big enough even for very large and complex data models. However this comes at a cost in memory (mainly stack) usage - the size of a confd_hkeypath_t is approximately 5.5 kB. Also, in some rare cases, we may have a data model where one or both of these values are not large enough.

It is possible to use other values for `MAXDEPTH` and `MAXKEYLEN`, but this requires both that `libconfd` is rebuilt from source with the new values, and that all applications that use `libconfd` are also compiled with the new values. It is of course possible to just edit `confd_lib.h` with the new values, but the `#define` statements for these in `confd_lib.h` are guarded with `#ifndef` directives, which means that they can alternatively be overridden without changing `confd_lib.h`.

Overriding can be done either via `-D` options on the compiler command line, or via `#define` statements before the `#include` for `confd_lib.h`. For building `libconfd` itself without source changes, only the `-D` option method is possible, though. The build procedure supports an `EXTRA_CFLAGS` **make** variable that can be used this purpose, see the `README` file included in the libconfd source package. E.g. we can do the `libconfd` build with:

```
$ make EXTRA_CFLAGS="-DMAXDEPTH=10 -DMAXKEYLEN=5"
```

The `-D` option method can of course be used when building applications too, but it is probably less error-prone to use the `#define` method. E.g. if we make sure that none of the application C or C++ files include `confd_lib.h` (or `confd.h`) directly, but instead include say `app.h`, we can have this in `app.h`:

```
#define MAXDEPTH 10
#define MAXKEYLEN 5
#include <confd_lib.h>
```

Whenever an application connects to ConfD via one of the API functions (i.e. `confd_connect()`, `cdb_connect()`, etc), a check is made that the `MAXDEPTH` and `MAXKEYLEN` values used for building the library are large enough for the data models loaded into ConfD. If they are not, the connection will fail with `confd_errno` set to `CONFD_ERR_PROTOUSAGE` and `confd_lasterr()` giving a message with the required minimum values. Whether the connection succeeds or not, the library will also set the global variables `confd_maxdepth` and `confd_maxkeylen` to the minimum values required by ConfD. Thus the values can be found by simply printing these variables in any application that connects to ConfD.

27.14. Error Message Customization

The ConfD release includes a XML document, `$CONFD_DIR/src/confd/errors/errcode.xml`, that specifies all the customizable errors that may be reported in the different northbound interfaces. The errors are classified with a type and a code, and for each error a parameterized format string for the default error message is given.

The purpose of this file is both to serve as a reference list of the possible errors, which could e.g. be processed programmatically when generating end-user documentation, and to provide the basis for error message customization.

All the error messages specified in the file can be customized by means of application callbacks. An application can register a callback for one or more of the error types, and whenever an error is to be reported in a northbound interface, the callback will first be invoked and given the opportunity to return a message that is different from the default.

The callback will receive user session information, the error type and code, the default error message, and the parameters used to create the default message. For errors of type "validation", the callback also

has access to the contents of the transaction that failed validation. See the section called “ERROR FORMATTING CALLBACK” in the confd_lib_dp(3) manual page for the details of the callback registration and invocation.

27.15. Using a different version of OpenSSL

ConfD depends on the OpenSSL `libcrypto` shared library for a number of cryptographic functions. (The `libssl` library is not used by ConfD.) Currently most ConfD releases, in particular all releases for Linux systems, are built with OpenSSL version 1.0.0, and thus require that the `libcrypto` library from this version is present when ConfD is run. Some releases for other systems require `libcrypto` from OpenSSL version 0.9.8 or 0.9.7. It is also possible that a given version, even though it is the one that ConfD requires, has been built with configuration parameters that make the interface incompatible with the build that is expected by ConfD.

However the `libcrypto` dependency is limited to two components in the ConfD release, the `libconfd` library used by applications, and a shared object called `crypto.so`, that is used by the ConfD daemon as an interface to `libcrypto`. Both these components are included in source form in the `confd-<vsn>.libconfd.tar.gz` tar archive that is provided with each ConfD release.

To use a different OpenSSL version than the one the ConfD release is built with, e.g. due to a Linux development or target environment having OpenSSL version 0.9.8 installed for other purposes, it is sufficient to use the provided sources to rebuild these two components with the desired OpenSSL version, and replace them in the ConfD release. The toplevel README file included in the tar archive has instructions on how to do the build of both `libconfd` and `crypto.so`.

While `libconfd` can be located wherever it is convenient for application use, `crypto.so` *must* be placed in the `$CONFD_DIR/lib/confd/lib/core/crypto/priv/lib` directory in the ConfD installation. The Makefiles in the tar archive have `install` targets for `libconfd` and `crypto.so` that will do a copy to the appropriate place in the ConfD installation if `CONFD_DIR` is set to the installation directory.

27.16. Using shared memory for schema information

It is possible to use shared memory to make schema information (see the section called “USING SCHEMA INFORMATION” in confd_types(3)) available to multiple processes on a given host, without requiring each of them to load the information directly from ConfD by calling one of the schema-loading functions (`confd_load_schemas()` etc, see the confd_lib_lib(3) and confd_lib_maapi(3) manual pages). This can be a very significant performance improvement for system startup, where multiple application processes will otherwise load schema information more or less simultaneously, and can also reduce RAM usage.

The mechanism uses a shared memory mapping created by `mmap(2)`, backed by a file. One process needs to call first `confd_mmap_schemas_setup()`, and then one of schema-loading functions, to populate the shared memory segment. Once this has been done, any process (including the one doing the initial load) can call `confd_mmap_schemas()` to map the shared memory segment into its address space and make the information available to the `libconfd` library and for direct access by the application. See the confd_lib_lib(3) manual page for the specification of these functions.

The mechanism can be used in different ways, but assuming that persistent storage for the backing file is available, the optimal approach is to do the load and file creation step only on first system start and when a data model upgrade is done. Then it is sufficient to call `confd_mmap_schemas()` on all other occasions. If persistent storage is not available, a RAM-based file system such as Linux "tmpfs" can be

used for the backing file, in which case the load and file creation step needs to be done on each boot (and on data model upgrade).

Since the schema information includes absolute pointers (e.g. the `parent`, `children`, and `next` pointers in a struct confd_cs_node), it is necessary to map the shared memory at the same virtual address in all processes. The *`addr`* argument to `confd_mmap_schemas_setup()` is passed to `mmap(2)`, and the address returned by `mmap(2)` is used for the mapping. The address is also recorded in the shared memory segment to make it available for `confd_mmap_schemas()`. The value of the *`size`* argument is also passed in the initial `mmap(2)` invocation, unless it is smaller than the first allocation done (e.g. if it is 0). In any case, unless the `CONFD_MMAP_SCHEMAS_KEEP_SIZE` flag is passed to `confd_mmap_schemas_setup()`, the loading will extend the mapped segment as needed, and the final size will only be as large as needed for the data, even if a larger value was passed as *`size`*.

Ideally we would give NULL for the *`addr`* argument and an approximate size for *`size`*, letting the kernel choose a suitable address and letting the load step adjust the final size based on the amount of data loaded. Unfortunately this often results in an address that is not honored on the subsequent `mmap(2)` call done by `confd_mmap_schemas()`, which thus fails. The possible choices of *`addr`* and/or *`size`* to get the desired result are OS- and OS-version-dependant, but on Linux it generally works to use an *`addr`* argument that is at an offset from the top of the heap that is larger than expected heap usage, and give *`size`* as 0, as shown in the sample code below using a 256 MB offset. (It is not a fatal error if heap usage later exceeds this offset, as `malloc(3)` etc will skip over the mapped area, but it may have some performance impact.)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <confd_lib.h>

#define MB (1024 * 1024)
#define SCHEMA_FILE "/etc/schemas"

#define OK(E) do {                                                      \
        int _ret = (E);                                                 \
        if (_ret != CONFD_OK) {                                         \
            confd_fatal(                                                \
                "%s returned %d, confd_errno=%d, confd_lasterr()='%s'\n", \
                #E, _ret, confd_errno, confd_lasterr());                \
        }                                                               \
    } while (0)

static void *get_shm_addr(size_t offset)
{
    size_t pagesize;
    char *addr;

    pagesize = (size_t)sysconf(_SC_PAGESIZE);
    addr = malloc(1);
    free(addr);
    addr += offset;
    /* return pagesize-aligned address */
    return addr - ((uintptr_t)addr % pagesize);
```

```
}

int main(int argc, char **argv)
{
    struct sockaddr_in addr;
    void *shm_addr;

    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);
    shm_addr = get_shm_addr(256 * MB);
    OK(confd_mmap_schemas_setup(shm_addr, 0, SCHEMA_FILE ".tmp", 0));
    OK(confd_load_schemas((struct sockaddr *)&addr,
                          sizeof(struct sockaddr_in)));
    if (rename(SCHEMA_FILE ".tmp", SCHEMA_FILE) != 0)
        confd_fatal("Failed to rename\n");
    return 0;
}
```

This code uses a temporary file that is renamed after the load is complete. This is not necessary, but ensures that the SCHEMA_FILE always represents complete schema info if it exists. It can also serve as a simple synchronization mechanism to let other processes know when they can do their `confd_mmap_schemas()` call.

On Solaris (at least Solaris 10), the address passed to `mmap(2)` is effectively ignored, and the returned address depends strictly on the size of the mapping. Thus there is no point passing anything other than NULL for the *`addr`* to `confd_mmap_schemas_setup()`, but instead the *`size`* must be big enough for the loaded schema info, and the `CONFD_MMAP_SCHEMAS_KEEP_SIZE` flag must be used.

In a multi-node system, with application processes connecting to ConfD across a network, shared memory can of course not be used between the nodes. The most straightforward way to handle this is to do the initial load and file creation step on each node. If the nodes have the same HW architecture and OS, a possible alternative could be to copy the backing store file from one node to the others using some file transfer mechanism.

27.17. Running application code inside ConfD

27.17.1. The econfd API

The Erlang API to ConfD is implemented as an Erlang/OTP application called `econfd`. It is included in source form in the ConfD release, in the `$CONFD_DIR/erlang` directory. Building `econfd` as described in the `$CONFD_DIR/erlang/econfd/README` file will compile the Erlang code and generate the documentation.

This API can be used by applications written in Erlang in much the same way as the C and Java APIs are used, i.e. code running in an Erlang VM can use the `econfd` API functions to make socket connections to ConfD for data provider, MAAPI, CDB, etc access. However the API is also available internally in ConfD, which makes it possible to run Erlang application code inside the ConfD daemon, without the overhead imposed by the socket communication.

There is little or no support for testing and debugging Erlang code executing internally in ConfD, since ConfD provides a very limited runtime environment for Erlang in order to minimize disk and memory footprints. Thus the recommended method is to develop Erlang code targeted for this by using `econfd` in

a separate Erlang VM, where an interactive Erlang shell and all the other development support included in the the standard Erlang/OTP releases are available. When development and testing is completed, the code can be deployed to run internally in ConfD without changes.

For information about the Erlang programming language and development tools, please refer to www.erlang.org and the available books about Erlang (some are referenced on the web site).

27.17.2. Running inside ConfD

All application code SHOULD use the prefix "ec_" for module names, registered processes (if any), and named ets tables (if any), to avoid conflict with existing or future names used by ConfD itself.

When ConfD starts, specifically when phase0 is reached, ConfD will search the load path as defined by `/confdConfig/loadPath` for compiled Erlang modules, i.e. `*.beam` files. The modules that are found will be loaded, unless the module name conflicts with an existing ConfD module. If there is a module name conflict, ConfD will terminate with an error message and exit code 21.

The application modules should use the `-on_load()` directive to spawn a process that makes initial connections to the ConfD subsystems, registers callbacks, sets up supervision if desired, etc. While the internal connections are made using the exact same API functions (e.g. `econfd_maapi:connect/2`) as for an application running in an external Erlang VM, any *`Address`* and *`Port`* arguments are ignored, and instead standard Erlang inter-process communication is used. The `internal_econfd/transform` example in the bundled collection shows a transform written in Erlang and executing internally in ConfD.

The `--printlog` option to **confd**, which prints the contents of the ConfD errorLog, is normally only useful for Tail-f support and developers, but it may also be relevant for debugging problems with application code running inside ConfD. The errorLog collects the events sent to the OTP error_logger, e.g. crash reports as well as info generated by calls to functions in the error_logger(3) module. Another possibility for primitive debugging is to run **confd** with the `--foreground` option, where calls to `io:format/2` etc will print to standard output.

While Erlang application code running in an external Erlang VM can use basically any version of Erlang/OTP, this is not the case for code running inside ConfD, since the Erlang VM is evolving and provides limited backward/forward compatibility. In general, running code compiled for a newer version in an older version of the VM is not supported (it may fail to load), whereas running code compiled for a version that is up to two major versions older than the version of the VM is supported (versions older than that may also work). The version of Erlang/OTP provided by ConfD in a given release can be found in the `VERSION` file in the toplevel directory of the installation. ConfD provides the VM and the `kernel`, `stdlib`, and `crypto` OTP applications.

Note

Obviously application code running internally in the ConfD daemon can have an impact on the execution of the standard ConfD code. Thus it is critically important that the application code is thoroughly tested and verified before being deployed for production in a system using ConfD.

27.17.3. User-defined types

We can implement user-defined types with Erlang code in a manner similar to what is described for C in the section called “USER-DEFINED TYPES” in confd_types(3). In the econfd API, we populate a `#confd_type_cbs{}` record and register it using `econfd_schema:register_type_cbs/1`. For an application running inside ConfD, this registration will have the same effect as using a shared object in the C API, i.e. the callback functions will be used internally by ConfD for doing string <-> value translation and syntax validation.

Callbacks for user-defined types may in general be required to be registered very early in the ConfD startup, in particular default values specified in the YANG data model will be translated from string form to internal representation when the corresponding `.fxs` file is loaded. With the shared object method used for C, this is always fulfilled, but for Erlang callbacks we need to take this into consideration, since the general loading of `*.beam` files when phase0 is reached, as described in the previous section, is too late for the translation of default values for user-defined types. By giving a module implementing such callbacks a name starting with "ec_user_type" (i.e. file name `ec_user_type*.beam`), we can tell ConfD that it should be loaded early enough for default value translation. Conversely, we should not have e.g. registration of normal data provider callbacks in such a module, since ConfD is not prepared to handle such registrations at this early point in the startup.

The `internal_econfd/user_type` example shows how the callbacks can be implemented in Erlang. It uses this naming convention to be able to handle the translation of a default value specified in the data model.

ConfD man-pages, Volume 1

Table of Contents

Name

confd — command to start and control the ConfD daemon

Synopsis

`confd` [--conf *`ConfFile`*] [--cd *`Dir`*] [--libdir *`LibDir`*] [--addloadpath *`Dir`*] [--nolog] [--smp *`Nr`*] [--foreground [-v | --verbose] [--stop-on-eof]] [--ignore-initial-validation] [--full-upgrade-validation] [--start-phase0] [--epoll { true | false }]

`confd` { --wait-phase0[*`TryTime`*] | --start-phase1 | --start-phase2 | --wait-started[*`TryTime`*] | --clear-aaa-cache | --reload | --areload | --status | --check-callbacks [*`Namespace`* | *`Path`*] | --loadfile *`File`* | --rollback *`Nr`* | --debug-dump *`File`* | --cli-j-dump *`File`* | --cli-i-dump *`File`* | --cli-c-dump *`File`* | --cli-check-templates | --loadxmlfiles *`File`*... | --mergexmlfiles *`File`*... | --cdb-backup *`File`* | --stop } [--timeout *`MaxTime`*]

`confd` { --version | --cdb-debug-dump *`Directory`* | --printlog *`BaseFileName`* | --set-snmp-engine-boots *`Nr`* }

DESCRIPTION

Use this command to start and control the ConfD daemon.

STARTING CONFD

These options are relevant when starting the ConfD daemon.

`-c`, `--conf` ConfFile	ConfFile is the path to a confd.conf file. The default location is defined when ConfD is installed, typically /etc/confd/confd.conf.
`--cd` Dir	Change working directory
`-l`, `--libdir` LibDir	LibDir is where the ConfD helper libraries are found. The default location is defined when ConfD is installed. The use of this flag is deprecated since confd figures out this information automatically, and using the confd command with LibDir from a different version of ConfD is not supported.
`--addloadpath` Dir	Add Dir to the set of directories ConfD uses to load fxs, clispec and, optionally, bin files.
`--nolog`	Do not log initial startup messages to syslog.
`--smp` Nr	Number of threads to run for Symmetric Multiprocessing (SMP). The default is 1, i.e. no SMP support. A value bigger than 1 will enable SMP support, where ConfD will at any given time use at most as many logical processors as the number of threads. The SMP behavior can also be affected by removing one of the two ConfD daemon executables from the installation, see the section Installing ConfD on a target system in the Advanced Topics chapter in the User Guide for the details of this.
`--foreground [ -v | --verbose ] [ --stop-on-eof ]`	Do not start as a daemon. Can be used to start ConfD from a process manager. In combination with -v or --verbose, all log messages are printed to stdout. Useful during development. In combination with

--stop-on-eof, ConfD will stop if it receives EOF (ctrl-d) on standard input. Note that to stop ConfD when run in foreground, send EOF (if --stop-on-eof was used) or use confd --stop. Do not terminate with ctrl-c, since ConfD in that case won't have the chance to close the database files.

`--ignore-initial-validation` When CDB starts on an empty database, or when upgrading, it starts a transaction to load the initial configuration or perform the upgrade. This option makes ConfD skip any validation callpoints when committing these initial transaction. (The preferred alternative is to use start-phases and register the validation callpoints in phase 0, see the user guide).

`--full-upgrade-validation` Perform a full validation of the entire database if the data models have been upgraded. This is useful in order to trigger external validation to run even if the database content has not been modified.

`--start-phase0` Start the daemon, but only start internal subsystems and CDB. Phase 0 is used when a controlled upgrade is done.

`--epoll { true | false }` Determines whether ConfD should use an enhanced poll() function (e.g. Linux epoll(7)). This can improve performance when ConfD has a high number of connections, but there may be issues with the implementation in some OS/kernel versions. The default is false.

COMMUNICATING WITH CONFD

When the ConfD daemon has been started, these options are used to communicate with the running daemon.

By default these options will perform their function by connecting to a running ConfD daemon over the loopback interface on the standard port. If the environment variable `CONFD_IPC_PORT` and/or `CONFD_IPC_ADDR` are set then the address/port in those variables will be used to communicate with the ConfD daemon. (Must be used if the daemon is not listening on its standard port on localhost, see the `/confdConfig/confdIpcAddress/ip` setting in the confd.conf(5) man-page, and the section on ConfD IPC in the ConfD Users Guide).

`--wait-phase0 [ TryTime ]` This call hangs until ConfD has initialized start phase0. After this call has returned, it is safe to register validation callbacks, upgrade CDB etc. This function is useful when ConfD has been started with --foreground and --start-phase0. It will keep trying the initial connection to ConfD for at most TryTime seconds (default 5).

For an equivalent C function see `maapi_wait_start()` in confd_lib_maapi(3).

`--start-phase1` Do not start the subsystems that listen to the management IP address. Must be called after the daemon was started with --start-phase0.

For an equivalent C function see `maapi_start_phase()` in confd_lib_maapi(3).

`--start-phase2` Must be called after the management interface has been brought up, if --start-phase1 has been used. Starts the subsystems that listens to the management IP address.

For an equivalent C function see `maapi_start_phase()` in confd_lib_maapi(3).

`--wait-started [ TryTime ]`

This call hangs until ConfD is completely started. This function is useful when ConfD has been started with --foreground. It will keep trying the initial connection to ConfD for at most TryTime seconds (default 5).

For an equivalent C function see `maapi_wait_start()` in confd_lib_maapi(3).

`--clear-aaa-cache`

Clear the ConfD AAA cache. When the AAA namespace is stored outside ConfD, for example through the *confd_aaa_bridge.c* program, ConfD must be notified when there is new AAA data to be read. ConfD caches all AAA data and this command will force ConfD to reload the AAA cache.

For an equivalent C function see `maapi_aaa_reload()` in confd_lib_maapi(3).

`--reload`

Reload the ConfD daemon configuration. All log files are closed and reopened, which means that **confd --reload** can be used from e.g. logrotate(8) - however it is preferable to use `maapi_reopen_logs()` for this, see confd_lib_maapi(3). `maapi_reopen_logs()` can also be invoked via **confd_cmd -c reopen_logs**, see confd_cmd(1).

For an equivalent C function see `maapi_reload_config()` in confd_lib_maapi(3).

Note

If we update a .fxs file, it is not enough to do a reload; the daemon has to be restarted, or the procedure described in the In-service Data Model Upgrade chapter in the User Guide has to be used.

`--areload`

Asynchronously reload the ConfD daemon configuration. This can be used in scripts executed by the ConfD daemon.

For an equivalent C function see `maapi_reload_config()` in confd_lib_maapi(3).

`--stop`

Stop the ConfD daemon.

For an equivalent C function see `maapi_stop()` in confd_lib_maapi(3).

`--status`

Prints status information about the ConfD daemon on stdout. Among the things listed are: loaded namespaces, current user sessions, callpoints (and whether they are registered or not), CDB status, and the current start-phase. Start phases are reported as "status:" and can be one of starting (which is pre-phase0), phase0, phase1, started (i.e. phase2), or stopping (which means that ConfD is about to shutdown).

`--debug-dump File` — Dump debug information from an already running ConfD daemon into a file. The file only makes sense to ConfD developers. It is often a good idea to include a debug dump in ConfD trouble reports.

`--cli-j-dump File` — Dump cli structure information from the ConfD daemon into a file.

`--cli-i-dump File` — Dump cli structure information from the ConfD daemon into a file.

`--cli-c-dump File` — Dump cli structure information from the ConfD daemon into a file.

`--cli-check-templates` — Walks through the entire data tree and validates all templates and verifies that all paths in the templates are valid.

`--check-callbacks [Namespace | Path]` — Walks through the entire data tree (config and stat), or only the Namespace or Path, and verifies that all read-callbacks are implemented for all elements, and verifies their return values.

`--loadfile File` — Load configuration in curly bracket format from File.

`--rollback Nr` — Rollback configuration to saved configuration number Nr.

`--loadxmlfiles File ...` — Load configuration in XML format from Files. The configuration is completely replaced by the contents in Files.

`--mergexmlfiles File ...` — Load configuration in XML format from Files. The configuration is merged with the contents in Files. The XML may use the 'operation' attribute, in the same way as it is used in a NETCONF <edit-config> operation.

`--cdb-backup File` — Save a snapshot of the CDB database into a GZipp:ed tar archive file (given by the File argument). If the File argument is a relative path, the file will be saved relative the *ConfD daemon's current working directory*. Only configuration data stored in CDB is saved (persistent CDB operational data is not). Note: if the database is locked for writing, the command will fail.

`--timeout MaxTime` — Specify the maximum time to wait for the ConfD daemon to complete the command, in seconds. If this option is not given, no timeout is used.

STANDALONE OPTIONS

`--cdb-debug-dump Directory` — Print a lot of information about the CDB files in *`Directory`* to stdout. This is a completely stand-alone feature and the only thing needed is the .cdb files (no running ConfD daemon or .fxs files etc).

`--version` — Reports the ConfD version without interacting with the daemon.

`--printlog BaseFileName` — Print the contents of the ConfD errorLog. This is normally only useful for Tail-f support and developers, since the information pertains to internal details of the ConfD software components, but it may also be relevant for Erlang application code executed internally in ConfD. The argument is the name as specified by `/confdConfig/logs/errorLog/filename`, i.e. without the `.idx`, `.1` etc suffixes.

`--set-snmp-engine-boots Nr` — Set the initial value, or override the previous value for the snmpEngineBoots counter object. After invoking ConfD with this option and (re-)starting ConfD the counter's value will be *Nr* + 1. This is potentially useful if an SNMP Agent implementation is being replaced by ConfD, using the same snmpEngineId as the previous agent implementation, and not wanting to clear the snmpEngineBoots counter for SNMP managers that have been communicating with the old Agent using SNMP v3. Invoking ConfD with this option must be done with `/confdConfig/stateDir` as working directory.

`--timeout MaxTime` — See above.

DIAGNOSTICS

If ConfD starts, the exit status is 0. If not it is a positive integer. The different meanings of the different exit codes are documented in the Advanced topics chapter in the user guide. When failing to start, the reason is normally given in the ConfD daemon log. The location of the daemon log is specified in the ConfFile as described in confd.conf(5).

SEE ALSO

confd.conf(5) - ConfD daemon configuration file format

Name

confd_aaa_bridge — Populating ConfD aaa_bridge.fxs with external data

Synopsis

```
<aaa>
    <aaaBridge>
      <enabled>true</enabled>
      <file>/etc/confd/aaa.conf</file>
    </aaaBridge>
  </aaa>
```

DESCRIPTION

Note

This program is deprecated. It does not support the NACM data model for access control.

ConfD needs to have the YANG module defining the namespace `http://tail-f.com/ns/aaa/1.1` defined. The namespace is mandatory for ConfD to run. The namespace specifies authentication and authorization data for ConfD and ConfD doesn't run unless this namespace is populated. This is fully described in the document "The ConfD AAA infrastructure"

We can either choose to use CDB to populate the AAA namespace in which case no C code needs to be written. Using CDB is the easiest and recommended way to populate the AAA. In the CDB case we should choose to have the "aaa_cdb.fxs" file in ConfD load path. By default ConfD use CDB to store the AAA data, thus this man page is only of interest for users that don't use CDB to store the "/aaa" tree.

If we do not want to use CDB, we can choose to populate the namespace using "aaa_bridge.fxs" using external data in which case a program - using the ConfD external data API from libconfd.so must be written to populate aaa_bridge.fxs.

confd_aaa_bridge.c is an example of such a program. It reads and writes an ad hoc .ini file which is used as "external database" for authentication and authorization data. If we enable confd_aaa_bridge in the configuration file for ConfD (see confd.conf(5)) ConfD will automatically start a precompiled version of confd_aaa_bridge on startup and stop it on shutdown.

Note

confd_aaa_bridge is just an example of how we can choose to populate the AAA namespace if we do not want to use CDB at all.

confd_aaa_bridge reads and writes a file with the following syntax:

```
[users]
 .. a set of users

[groups]
 .. a set of groups

[cmdrules]
 .. a set of rules

[datarules]
 .. a set of rules
```

The [users] are specified as six space/tab separated fields

```
  user    uid   gid cryptpassword   sshdir  homedir
```

The user field is the name of the user, the cryptpassword is the encrypted (see man crypt(3)) password of the user. The sshdir is the name of a directory where the users SSH keys are kept and finally the homedir is a directory which is considered the HOME directory of the user. The CLI will save files in this directory. The uid and gid are UNIX ids ConfD will use to run commands on behalf of the logged in user.

```
[users]
admin 0 0 $1$feedbabe$nGlMYlZpQ0bzenyFOQI3L1 /var/u1/.ssh /var/u1
oper  0 0 $1$feedbabe$i2glnaB.iUj2VXh/zlq.o/ /var/u2/.ssh /var/u2
```

The [groups] are specified as several space/tab separated fields

```
group   gid user1 user2 ......
```

The first field, the *group* is the name of a group, the remainder of space separated strings is a list of users being members in the group. The gid is the UNIX group id of this group. -1 means that no additional group id should be assigned to a user that belongs to this group.

The [cmdrules] are specified as six space/tab separated fields:

```
index context command group op action
```

And the [datarules] are specified of seven space/tab separated fields

```
index  context  namespace  keypath  group  op  action
```

The meaning of the different rule fields is described in the AAA userguide.

SIGNALS

If the signal SIGHUP is sent to the program as in

```
# killall -HUP confd_aaa_bridge
```

The program will die, ConfD will notice the exit code and silently restart confd_aaa_bridge. This is a convenient way to force ConfD to reload a data file edited by hand, is to kill -HUP the confd_aaa_bridge UNIX process

SEE ALSO

See the YANG module tailf-aaa.yang in the $CONFD_DIR/src/confd/aaa directory in the release, as well as the accompanying annotation YANG module bridge-ann.yang in the $CONFD_DIR/src/confd/confd_aaa_bridge directory, which brings the necessary callpoint into the original YANG module tailf-aaa.yang.

Name

confd_cli — Frontend to the ConfD CLI engine

Synopsis

`confd_cli` [*`options`*] [*`File`*]

`confd_cli` [--help] [--host *`Host`*] [--ip *`IpAddress`* | *`IpAddress/Port`*] [--address *`Address`*] [--port *`PortNumber`*] [--cwd *`Directory`*] [--proto `tcp` | `ssh` | `console`] [--interactive] [--noninteractive] [-J | -C | -I] [--user *`Username`*] [--uid *`UidInt`*] [--groups *`Groups`*] [--gids *`GidList`*] [--gid *`Gid`*] [--opaque *`Opaque`*] [--noaaa]

DESCRIPTION

The confd_cli program is a C frontend to the ConfD CLI engine. The **confd_cli** program connects to ConfD and basically passes data back and forth from the user to ConfD.

confd_cli can be invoked from the command line. If so, no authentication is done. The archetypical usage of confd_cli is to use it as a login shell in /etc/passwd, in which case authentication is done by the login program.

The source code for **confd_cli** resides in `$CONFD_DIR/src/confd/cli` and can be modified if required.

OPTIONS

`-h`, `--help`	Display help text.
`-H`, `--host` *`HostName`*	Gives the name of the current host. The **confd_cli** program will use the value of the system call `gethostbyname()` by default. The host name is used in the CLI prompt.
`-i`, `--ip` *`IpAddress`* | *`IpAddress/Port`*	Set the IP (or IP address and port) which ConfD reports that the user is coming from. The **confd_cli** program by default tries to determine this automatically by reading the `SSH_CONNECTION` environment variable.
`-A`, `--address` *`Address`*	CLI address to connect to. The default is 127.0.0.1. This can be controlled by either this flag, or the UNIX environment variable `CONFD_IPC_ADDR`. The `-A` flag takes precedence.
`-P`, `--port` *`PortNumber`*	CLI port to connect to. The default is the ConfD IPC port, which is 4565 This can be controlled by either this flag, or the UNIX environment variable `CONFD_IPC_PORT`. The `-P` flag takes precedence.
`-c`, `--cwd` *`Directory`*	The current working directory for the user once in the CLI. All file references from the CLI will be relative to the cwd. By default the value will be the actual cwd where confd_cli is invoked.
`-p`, `--proto` `ssh` | `tcp` | `console`	The protocol the user is using. If `SSH_CONNECTION` is set, this defaults to "ssh", otherwise "console".

`-n`, `--interactive`	This forces the CLI to run in interactive mode. In non interactive mode, the CLI never prompts the user for any input. This flag can sometimes be useful in certain CLI scripting scenarios.
`-N`, `--noninteractive`	This forces the CLI to run in non interactive mode. See Section 16.4.1, “Starting the CLI” for further info.
`-J`, `-C`, `-I`	This flag sets the mode of the CLI. `-J` is Juniper style CLI, `-C` is Cisco XR style CLI and `-I` is Cisco IOS style CLI.
`-u`, `--user` *`User`*	Indicates to ConfD which username the user has. This defaults to the username of the invoker.
`-U`, `--uid` *`Uid`*	Indicates to ConfD which uid the user has.
`-g`, `--groups` *`GroupList`*	Indicates to ConfD which groups the user are a member of. The parameter is a comma separated string. This defaults to the actual UNIX groups the user is a member of. The group names are used by the AAA system in ConfD to authorize data and command access.
`-D`, `--gids` *`GidList`*	Indicates to ConfD which secondary group ids the user shall have. The parameter is a comma separated string of integers. This defaults to the actual secondary UNIX group ids the user has. The gids are used by ConfD when ConfD executes commands on behalf of the user.
`-G`, `--gid` *`Gid`*	Indicates to ConfD which group id the user shall have. This defaults to the actual UNIX group id the user has. The gid is used by ConfD when ConfD executes commands on behalf of the user.
`-O`, `--opaque` *`Opaque`*	Pass an opaque string to ConfD. The string is not interpreted by ConfD, only made available to application code. See "built-in variables" in clispec(5) and `maapi_get_user_session_opaque()` in confd_lib_maapi(3). The string can be given either via this flag, or via the UNIX environment variable `CONFD_CLI_OPAQUE`. The `-O` flag takes precedence.
`--noaaa`	Completely disables all AAA checks for this CLI. This can be used as a disaster recovery mechanism if the AAA rules in ConfD have somehow become corrupted.

ENVIRONMENT VARIABLES

CONFD_IPC_ADDR	Which IP address to connect to.
CONFD_IPC_PORT	Which TCP port to connect to.
SSH_CONNECTION	Set by openssh and used by *confd_cli* to determine client IP address etc.
TERM	Passed on to terminal aware programs invoked by ConfD.

EXIT CODES

0 Normal exit

1 Failed to read user data for initial handshake.

2 Close timeout, client side closed, session inactive.

3 Idle timeout triggered.

4 Tcp level error detected on daemon side.

5 Internal error occurred in daemon.

5 User interrupted clistart using special escape char.

6 User interrupted clistart using special escape char.

7 Daemon abruptly closed socket.

SCRIPTING

It is very easy to use **confd_cli** from **/bin/sh** scripts. **confd_cli** reads stdin and can then also be run in non interactive mode. This is the default if stdin is not a tty (as reported by `isatty()`)

Here is example of invoking **confd_cli** from a shell script.

```
#!/bin/sh

confd_cli << EOF
configure
set foo bar 13
set funky stuff 44
commit
exit no-confirm
exit
EOF
```

And here is en example capturing the output of **confd_cli**:

```
#!/bin/sh
{ confd_cli << EOF;
configure
set trap-manager t2 ip-address 10.0.0.1 port 162 snmp-version 2
commit
exit no-confirm
exit
EOF
} | grep 'Aborted:.*not unique.*'
if [ $? != 0 ]; then
  echo 'test2: commit did not fail'; exit 1;
fi
```

The above type of CLI scripting is a very efficient and easy way to test various aspects of the CLI.

Name

confd_cmd — Command line utility that interfaces to common ConfD library functions

Synopsis

`confd_cmd` [*(1) options*] [*`filename`*]

`confd_cmd` [*(1) options*] -c *`string`*

`confd_cmd` -h | -h commands | -h *`command-name`*...
(1) [-r | -o | -e | -S] [-f [w] | [p] [r | s]] [-a *`address`*] [-p *`port`*] [-u *`user`*] [-g *`group`*] [-x *`context`*] [-s] [-m] [-h] [-d]

DESCRIPTION

The **confd_cmd** utility is implemented as a wrapper around many common CDB and MAAPI function calls. The purpose is to make it easier to prototype and test various ConfD issues using normal scripting tools.

Input is provided as a file (default `stdin` unless a filename is given) or as directly on the command line using the `-c` *`string`* option. The **confd_cmd** expects commands separated by semicolon (;) or newlines. A pound (#) sign means that the rest of the line is treated as a comment. For example:

```
confd_cmd -c get_phase
```

Would print the current start-phase of ConfD, and:

```
confd_cmd -c "get_phase ; get_txid"
```

would first print the current start-phase, then the current transaction ID of CDB.

Sessions towards CDB, and transactions towards MAAPI are created as-needed. At the end of the script any open CDB sessions are closed, and any MAAPI read/write transactions are committed.

Source code to this utility is included in the distribution as `src/confd/tools/confd_cmd.c`.

OPTIONS

`-d` Debug flag. Add more to increase debug level. All debug output will be to stderr.

`-m` Don't load the schemas at startup.

ENVIRONMENT VARIABLES

`CONFD_IPC_ADDR`, `CONFD_IPC_EXTADDR`	The address used to connect to the ConfD daemon, overrides the compiled in default.
`CONFD_IPC_PORT`	The port number to connect to the ConfD daemon on, overrides the compiled in default.
`CONFD_IPC_EXTSOPATH`	The absolute path to the shared object to use for a connection using external IPC when `CONFD_IPC_EXTADDR` is given.

EXAMPLES

1. Getting the address of eth0

   ```
   confd_cmd -c "get /sys:sys/ifc{eth0}/ip"
   ```

2. Setting a leaf in CDB operational

   ```
   confd_cmd -o -c "set /sys:sys/ifc{eth0}/stat/tx 88"
   ```

3. Making ConfD running on localhost the master, with the name node0

   ```
   confd_cmd -c "master node0"
   ```

   Then tell the ConfD also running on localhost, but listening on port 4566, slave to the master. Calling the slave node1

   ```
   confd_cmd -p 4566 -c "slave node1 node0 127.0.0.1"
   ```

SEE ALSO

Source code, included in $`CONFD_DIR`/src/confd/tools/confd_cmd.c

confd_lib_maapi(3) - Confd MAAPI library

confd_lib_cdb(3) - Confd CDB library

Name

confd_load — Command line utility to load and save ConfD configurations

Synopsis

`confd_load` [-W] [-S] [*(1) common options*] [*`filename`*]

`confd_load` -l [-m | -r] [-D] [*(1) common options*] [*`filename`*...]

`confd_load` -C [-R] [*`filename`*...]

`confd_load` -h | -?
(1) [-d] [-t] [-F { x | p | o | j | c | i }] [-H | -U] [-a] [-e] [[-u *`user`*] [-g *`group`*...] [-c *`context`*] | [-i]] [[-p *`keypath`*] | [-P *`XPath`*]] [-o] [-s]

DESCRIPTION

The confd_load command is a command line interface to the functions `maapi_save_config()`, `maapi_load_config()`, `maapi_load_config_stream()`, and `cdb_load_file()` respectively.

This command provides a convenient way of loading and saving all or parts of the configuration in different formats. It can be used to initialize or restore configurations as well as in CLI commands.

If you run **confd_load** without any options it will print the current configuration in XML format on stdout. The exit status will be zero on success and non-zero otherwise.

Source code to this utility is included in the distribution as `src/confd/tools/confd_load.c`.

COMMON OPTIONS

`-d`	Debug flag. Add more to increase debug level. All debug output will be to stderr.
`-t`	Measure how long the requested command takes and print the result on stderr.
`-F` x | p | o | j | c | i	Selects the format of the configuration, must be set both when loading and saving. One of XML (x), pretty XML (p), JSON (o), curly braces J-style CLI (j), C-style CLI (c), or I-style CLI (i). Default is XML.
`-H`	Hide all hidden nodes. By default, no nodes are hidden unless **confd_load** has attached to an existing transaction, in which case the hidden nodes are the same as in that transaction's session.
`-U`	Unhide all hidden nodes. By default, no nodes are hidden unless **confd_load** has attached to an existing transaction, in which case the hidden nodes are the same as in that transaction's session.
`-u` *`user`*, `-g` *`group`* ..., `-c` *`context`*	Loading and saving the configuration is done in a user session, using these options it is possible to specify which user, groups (more than one `-g` can be used to add groups), and context that should be

used when starting the user session. If only a user is supplied the user is assumed to belong to a single group with the same name as the user. This is significant in that AAA rules will be applied for the specified user / groups / context combination. The default is to use the `system` context, which implies that AAA rules will *not* be applied at all.

Note

If the environment variables `CONFD_MAAPI_USID` and `CONFD_MAAPI_THANDLE` are set (see the ENVIRONMENT section), or if the `-i` option is used, these options are silently ignored, since **confd_load** will attach to an existing transaction.

`-i` Instead of starting a new user session and transaction, **confd_load** will try to attach to the init session. This is only valid when ConfD is in start phase 0, and will fail otherwise. It can be used to load a "factory default"file during startup, or loading a file during upgrade.

`-s` Start transaction towards the startup datastore (instead of the default running). I.e. when loading, the configuration loaded will be committed to startup, and as such won't take effect until ConfD is restarted.

SAVE CONFIGURATION

By default the complete current configuration will be output on stdout. To save it in a file add the filename on the command line (the `-f` option is deprecated). The file is opened by the **confd_load** utility, permissions and ownership will be determined by the user running **confd_load**. Output format is specified using the `-F` option.

When saving the configuration in XML format, the context of the user session (see the `-c` option) will determine which namespaces with export restriction (from `tailf:export`) that are included. If the `system` context is used (this is the default), all namespaces are saved, regardless of export restriction. When saving the configuration in one of the CLI formats, the context used for this selection is always `cli`.

A number of options are only applicable, or have a special meaning when saving the configuration:

`-f` *`filename`* Filename to save configuration to (option is deprecated, just give the filename on the command line).

`-W` Include leaves which are unset (set to their default value) in the output. By default these leaves are not included in the output. (Corresponds to the `MAAPI_CONFIG_WITH_DEFAULTS` flag).

`-S` Include the default value of a leaf as a comment (only works for CLI formats, not XML). (Corresponds to the `MAAPI_CONFIG_SHOW_DEFAULTS` flag).

`-p` *`keypath`* Only include the configuration below *`keypath`* in the output.

`-P` *`XPath`* Filter the configuration using the *`XPath`* expression. (Only works for the XML format.)

`-o` Include operational data in the output. (Corresponds to the `MAAPI_CONFIG_WITH_OPER` flag).

LOAD CONFIGURATION

When the `-l` option is present **confd_load** will load all the files listed on the command line using the `maapi_load_config()` function. The file(s) are expected to be in XML format unless otherwise specified using the `-F` flag. Note that it is the ConfD daemon that opens the file(s), it must have permission to do so. However relative pathnames are assumed to be relative to the working directory of the **confd_load** command (it will pass an absolute pathname to `maapi_load_config()`).

If neither of the `-m` and `-r` options are given when multiple files are listed on the command line, **confd_load** will silently treat the second and subsequent files as if `-m` had been given, i.e. it will merge in the contents of these files instead of deleting and replacing the configuration for each file. Note, we almost always want the merge behavior. If no file is given, or "-" is given as a filename, **confd_load** will stream standard input to ConfD by using `maapi_load_config_stream()`.

`-f` *`filename`* The file to load (deprecated, just list the file after the options instead).

`-m` Merge in the contents of *`filename`*, the (somewhat unfortunate) default is to delete and replace. (Corresponds to the `MAAPI_CONFIG_MERGE` flag).

`-x` Lax loading. Only applies to XML loading. Ignore unknown namespaces, attributes and elements.

`-r` Replace the part of the configuration that is present in *`filename`*, the default is to delete and replace. (Corresponds to the `MAAPI_CONFIG_REPLACE` flag).

`-a` When loading configuration in 'i' or 'c' format, do a commit operation after each line. Default and recommended is to only commit when all the configuration has been loaded. (Corresponds to the `MAAPI_CONFIG_AUTOCOMMIT` flag).

`-e` When loading configuration do not abort when encountering errors (corresponds to the `MAAPI_CONFIG_CONTINUE_ON_ERROR` flag).

`-D` Call `maapi_delete_all` (*`MAAPI_DEL_ALL`*) before loading the file.

`-p` *`keypath`* Call `maapi_delete(`*`keypath`*`)` before loading the file.

`-o` Accept but ignore contents in the file which is operational data (without this flag it will be an error). (Corresponds to the `MAAPI_CONFIG_WITH_OPER` flag)

LOAD CDB OPERATIONAL

The `-C` option is a direct interface to the `cdb_load_file()` function.

The `-C` option is used to load operational data. When you use `-C` all other options except `-R` (and `-d`) are ignored, since they don't apply. Files on the command line must be in XML format and will be fed to `cdb_load_file()` in the order they are listed. If no file is given, or "-" is given as a filename, **confd_load** will read standard input and use `cdb_load_str()` to load the collected data. If the `-R` option is included, CDB operational subscription notifications will be generated.

Any data which isn't part of CDB operational per the data model will be ignored. This means that you can save a single file with both configuration and operational data and feed it back to **confd_load**.

If you use a relative path for *`filename`* it is assumed to be relative to the working directory of the **confd_load** command (it will pass an absolute pathname to `cdb_load_file()`).

EXAMPLES

Example 138. Reloading all xml files in the cdb directory

```
confd_load -D -m -l cdb/*.xml
```

Example 139. Merging in the contents of `conf.cli`

```
confd_load -l -m -F j conf.cli
```

Example 140. Print interface config and statistics data in cli format

```
confd_load -F i -o -p /sys:sys/ifc
```

Example 141. Using xslt to format output

```
confd_load -F x -p /sys:sys/ifc | xsltproc fmtifc.xsl -
```

Example 142. Using xmllint to pretty print the xml output

```
confd_load -F x | xmllint --format -
```

Example 143. Saving config and operational data to `/tmp/conf.xml`

```
confd_load -F x -o > /tmp/conf.xml
```

Example 144. Restoring both config and operational data

```
confd_load -l -F x -o /tmp/conf.xml
confd_load -C /tmp/conf.xml
```

Example 145. Measure how long it takes to fetch config

```
confd_load -t > /dev/null
elapsed time: 0.011 s
```

Example 146. Output all instances in list `/foo/table` which has `ix` larger than 10

```
confd_load -F x -P "/foo/table[ix > 10]"
```

ENVIRONMENT

`CONFD_IPC_ADDR`, `CONFD_IPC_EXTADDR`	The address used to connect to the ConfD daemon, overrides the compiled in default.
`CONFD_IPC_PORT`	The port number to connect to the ConfD daemon on, overrides the compiled in default.
`CONFD_IPC_EXTSOPATH`	The absolute path to the shared object to use for a connection using external IPC when `CONFD_IPC_EXTADDR` is given.
`CONFD_MAAPI_USID`, `CONFD_MAAPI_THANDLE`	If set **confd_load** will use `maapi_attach2()` to attach to an existing transaction in an existing user session instead of starting a new session. These environment variables are set by the ConfD CLI when it invokes external commands, which means you can run **confd_load** directly from the

CLI. For example, the following addition to the `<operationalMode>` in a clispec file (see clispec(5))

```
<cmd name="servers" mount="show">
  <info/>
  <help/>
  <callback>
    <exec>
      <osCommand>confd_load</osCommand>
          <args>-F j -p /system/servers</args>
    </exec>
  </callback>
</cmd>
```

will add a **show servers** command which, when run will invoke **confd_load -F j -p /system/servers**. This will output the configuration below `/system/servers` in curly braces format.

Note that when these environment variables are set, it means that the configuration will be loaded into the current CLI transaction (which must be in configure mode, and have AAA permissions to actually modify the config). To load (or save) a file in a separate transaction, unset these two environment variables before invoking the **confd_load** command.

SEE ALSO

confd_lib_maapi(3) - Confd MAAPI library

confd_lib_cdb(3) - Confd CDB library

Name

confdc — Confdc compiler

Synopsis

confdc -c [-a | --annotate *YangAnnotationFile*] [--deviation *DeviationFile*] [-o *FxsFile*] [--verbose] [--fail-on-warnings] [-E | --error *ErrorCode*...] [-W | --warning *ErrorCode*...] [-w | --no-warning *ErrorCode*...] [--strict-yang] [--use-description [always]] [[--no-features] | [-F | --features *Features* ...]] [--ignore-unknown-features] [--subagent *MountPath*] [--yangpath *YangDir*] [--export *Agent* [-f *FxsFileOrDir*...] ...] -- *YangFile*

confdc --list-errors

confdc -c [-o *CclFile*] *ClispecFile*

confdc -c [-o *BinFile*] [-I *Dir*] *MibFile*

confdc -c [-o *BinFile*] [--read-only] [--verbose] [-I *Dir*] [--include-file *BinFile*] [--fail-on-warnings] [--warn-on-type-errors] [--warn-on-access-mismatch] [--mib-annotation *MibA*] [-f *FxsFileOrDir*...] -- *MibFile FxsFile*...

confdc --check-deps [-f *FxsFileOrDir*...] -- *FxsFile*...

confdc --emit-h *HFile* [--macro-prefix *Prefix*] [--include-type] [--exclude-enums] [--fail-on-warnings] *FxsFile*

confdc --emit-java *JFile* [--print-java-filename] [--java-disable-prefix] [--java-package *Package*] [--exclude-enums] [--fail-on-warnings] [-f *FxsFileOrDir* ...] *FxsFile*

confdc --emit-hrl *HrlFile* [--macro-prefix *Prefix*] [--include-type] [--exclude-enums] [--fail-on-warnings] *FxsFile*

confdc --emit-mib *MibFile* [--join-names *capitalize* | *hyphen*] [--oid *OID*] [--top *Name*] [--tagpath *Path*] [--import *Module Name*] [--module *Module*] [--generate-oids] [--generate-yang-annotation] [--skip-symlinks] [--top *Top*] [--fail-on-warnings] [--no-comments] [--read-only] -- *FxsFile*...

confdc --mib2yang [--mib-annotation *MibA*] [--emit-doc] [--snmp-name] [--read-only] [-u *Uri*] [-p *Prefix*] [-o *YangFile*] -- *MibFile*

confdc --snmpuser *EngineID User AuthType PrivType PassPhrase*

confdc --emit-xsd *XsdFile* [--xsd-import] [-f *FxsFileOrDir*...] -- *FxsFile*

confdc --revision-merge [-o *ResultFxs*] [-v] [-f *FxsFileOrDir*...] -- *ListOfFxsFiles*...

confdc --lax-revision-merge [-o *ResultFxs*] [-v] [-f *FxsFileOrDir*...] -- *ListOfFxsFiles*...

confdc --confm-emit-java *OutputDir* [--java-package *Package*] [--java-class-naming *Level*] [--java-class-naming-fixed] [--case-insensitive-naming] [-f *FxsFileOrDir*...] -- *FxsFile*

confdc --get-info *FxsFile*

confdc --get-uri *FxsFile*

confdc --get-version *FxsFile*

confdc --version

`confdc` --xmlcheck [-f *`FxsFile`*] -- *`XmlFile`*...

DESCRIPTION

During startup the ConfD daemon loads .fxs files describing our configuration data models. A .fxs file is the result of a compiled YANG data model file. The daemon also loads clispec files describing customizations to the auto-generated CLI. The clispec files are described in clispec(5).

A yang file by convention uses .yang (or .yin) filename suffix. YANG files are directly transformed into .fxs files by confdc.

We can use any number of .fxs files when working with the ConfD daemon.

Optionally a W3C XML Schema file (.xsd file) can be generated. This schema can be used to describe, using a standard notation, what actually goes over the wire. The `--emit-xsd` option is used to generate a .xsd file from a .fxs file.

The `--emit-h` option is used to generate a .h file from .fxs files. How to use the generated .h files are described in the ConfD User Guide.

The `--emit-java` option is used to generate a .java file from a .fxs file. The java file is used in combination with the Java library for Java based applications.

The `--emit-hrl` option is used to generate a .hrl file from .fxs files. The .hrl file can be used for Erlang based applications.

The `--print-java-filename` option is used to print the resulting name of the would be generated .java file.

The `--emit-mib` option is used to generate an SNMP MIB from .fxs files.

The `--snmpuser` option is used to generate localized keys for SNMP v3.

The `--confm-emit-java` option is used to generate data model aware java classes for the ConfM library. The classes are generated from an .fxs file.

A clispec file by convention uses a .cli filename suffix. We use the confdc command to compile a clispec into a loadable format (with a .ccl suffix).

A mib file by convention uses a .mib filename suffix. The confdc command is used for compiling the mib with one or more fxs files (containing OID to YANG mappings) into a loadable format (with a .bin suffix). See the ConfD User Guide for more information about compiling the mib.

Take a look at the EXAMPLE section for a crash course.

OPTIONS

Common options

`-f`, `--fxsdep` *`FxsFile-OrDir`*...	.fxs files (or directories containing .fxs files) to be used to resolve cross namespace dependencies.
`--yangpath` *`YangModuleDir`*	YangModuleDir is a directory containing other YANG modules and submodules. This flag must be used when we import or include other YANG modules or submodules that reside in another directory.

`-o`, `--output` *`File`*
: Put the resulting file in the location given by File.

Compile options

`-c`, `--compile` *`File`*
: Compile a YANG file (.yang/.yin) to a .fxs file or a clispec (.cli file) to a .ccl file, or a MIB (.mib file) to a .bin file

`-a`, `--annotate` *`AnnotationFile`*
: YANG users that are utilizing the tailf:annotate extension must use this flag to indicate the YANG annotation file(s).

  This parameter can be given multiple times.

`--deviation` *`DeviationFile`*
: Indicates that deviations from the module in *DeviationFile* should be present in the fxs file.

  This parameter can be given multiple times.

`-F`*`features`*, `--feature` *`features`*
: Indicates that support for the YANG *features* should be present in the fxs file. *`features`* is a string on the form *`modulename`*: [*`feature`*(,*`feature`*)*]

  This option is used to prune the data model by removing all nodes that are defined with a "if-feature" that is not listed as *`feature`*.

  This option can be given multiple times.

  If this option is not given, nothing is pruned, i.e., it works as if all features were explicitly listed.

  If the module uses a feature defined in an imported YANG module, it must be given as *`modulename:feature`*.

`--no-features`
: Indicates that no YANG features from the given module are supported.

`--ignore-unknown-features`
: Instructs the compiler to not give an error if an unknown feature is specified with `--feature`.

`--use-description [always]`
: Normally, 'description' statements are ignored by confdc. Instead the 'tailf:info' statement is used as help and information text in the CLI and Web UI. When this option is specified, text in 'description' statements is used if no 'tailf:info' statement is present. If the option *always* is given, 'description' is used even if 'tailf:info' is present.

`--export` *`Agent`* ...
: Makes the namespace visible to Agent. Agent is either "none", "all", "netconf", "snmp", "cli", "webui", "rest" or a free-text string. This option overrides any `tailf:export` statements in the module. The option "all" makes it visible to all agents. Use "none" to make it invisible to all agents.

`--subagent` *`MountPath`*
: This option is used to compile a subagent's YANG modules for the master agent. It tells the master agent that this namespace is handled by a subagent. MountPath is an XPath expression (without instance selectors) where the namespace is mounted in the master agent's data hierarchy.

`--fail-on-warnings`
: Make compilation fail on warnings.

`-W` *`ErrorCode`*

Treat *`ErrorCode`* as a warning, even if `--fail-on-warnings` is given. *`ErrorCode`* must be a warning or a minor error.

Use `--list-errors` to get a listing of all errors and warnings.

The following example treats all warnings except the warning for dependency mismatch as errors:

```
$ confdc -c --fail-on-warnings -W TAILF_DEPENDENCY_MISMATCH
```

`-w` *`ErrorCode`*

Do not report the warning *`ErrorCode`*, even if `--fail-on-warnings` is given. *`ErrorCode`* must be a warning.

Use `--list-errors` to get a listing of all errors and warnings.

The following example ignores the warning TAILF_DEPENDENCY_MISMATCH:

```
$ confdc -c -w TAILF_DEPENDENCY_MISMATCH
```

`-E` *`ErrorCode`*

Treat the warning *`ErrorCode`* as an error.

Use `--list-errors` to get a listing of all errors and warnings.

The following example treats only the warning for unused import as an error:

```
$ confdc -c -E UNUSED_IMPORT
```

`--strict-yang`

Force strict YANG compliance. Currently this checks that the deref() function is not used in XPath expressions and leafrefs.

MIB to YANG options

`--mib2yang` *`MibFile`*

Generate a YANG file from the MIB module (.mib file).

If the MIB IMPORTs other MIBs, these MIBs must be available (as .mib files) to the compiler when a YANG module is generated. By default, all MIBs in the current directory and all builtin MIBs are available. Since the compiler uses the tool **smidump** to perform the conversion to YANG, the environment variable `SMIPATH` can be set to a colon-separated list of directories to search for MIB files.

`-u`, `--uri` *`Uri`*

Specify a uri to use as namespace in the generated YANG module.

`-p`, `--prefix` *`Prefix`*

Specify a prefix to use in the generated YANG module.

`--mib-annotation` *`MibA`*

Provide a MIB annotation file to control how to translate specific MIB objects to YANG. See mib_annotations(5).

`--snmp-name`

Generate the YANG statement "tailf:snmp-name" instead of "tailf:snmp-oid".

`--read-only`

Generate a YANG module where all nodes are "config false".

MIB compiler options

`-c`, `--compile` *`MibFile`*

Compile a MIB module (.mib file) to a .bin file.

If the MIB IMPORTs other MIBs, these MIBs must be available (as compiled .bin files) to the compiler. By default, all compiled MIBs in the current directory and all builtin MIBs are available. Use the parameters *--include-dir* or *--include-file* to specify where the compiler can find the compiled MIBs.

`--verbose` Print extra debug info during compilation.

`--read-only` Compile the MIB as read-only. All SET attempts over SNMP will be rejected.

`-I`, `--include-dir` *`Dir`* Add the directory Dir to the list of directories to be searched for IMPORTed MIBs (.bin files).

`--include-file` *`File`* Add File to the list of files of IMPORTed (compiled) MIB files. File must be a .bin file.

`--fail-on-warnings` Make compilation fail on warnings.

`--warn-on-type-errors` Warn rather than give error on type checks performed by the MIB compiler.

`--warn-on-access-mis-match` Give a warning if an SNMP object has read only access to a config object.

`--mib-annotation` *`MibA`* Provide a MIB annotation file to fine-tune how specific MIB objects should behave in the SNMP agent. See mib_annotations(5).

Emit C header file options

`--emit-h` *`HFile`* Generate a .h utility header file to be used when working with the ConfD APIs.

`--macro-prefix` *`Prefix`* Without this option, all macro definitions in the generated .h file are prepended with the argument of the `prefix` statement in the YANG module. If this option is used, the macro definitions are prepended with Prefix instead.

`--include-type` If this option is used all macro definitions for enums in the generated .h file have the type name as part of their name.

`--exclude-enums` If this option is used, macro definitions for enums are omitted from the generated .h file. This can in some cases be useful to avoid conflicts between enum symbols, or between enums and other symbols.

`--fail-on-warnings` If this option is used all warnings are treated as errors and confdc will fail its execution.

Emit Erlang header file options

`--emit-hrl` *`HrlFile`* Generate a .hrl utility header file to be used when working with the ConfD Erlang APIs.

`--macro-prefix` *`Prefix`* Without this option, all macro definitions in the generated .hrl file are prepended with the argument of the `prefix` statement in the

YANG module. If this option is used, the macro definitions are prepended with Prefix instead.

`--include-type` — If this option is used all macro definitions for enums in the generated .hrl file have the type name as part of their name.

`--exclude-enums` — If this option is used, macro definitions for enums are omitted from the generated .hrl file. This can in some cases be useful to avoid conflicts between enum symbols, or between enums and other symbols.

`--fail-on-warnings` — If this option is used all warnings are treated as errors and confdc will fail its execution.

Emit SMIv2 MIB options

`--emit-mib` *`MibFile`* — Generates a MIB file for use with SNMP agents/managers. See the appropriate section in the SNMP agent chapter in the ConfD User Guide for more information.

`--join-names capitalize` — Join element names without separator, but capitalizing, to get the MIB name. This is the default.

`--join-names hyphen` — Join element names with hyphens to get the MIB name.

`--join-names force-capitalize` — The characters '.' and '_' can occur in YANG identifiers but not in SNMP identifiers; they are converted to hyphens, unless this option is given. In this case, such identifiers are capitalized (to lowerCamelCase).

`--oid` *`OID`* — Let *OID* be the top object's OID. If the first component of the OID is a name not defined in SNMPv2-SMI, the `--import` option is also needed in order to produce a valid MIB module, to import the name from the proper module. If this option is not given, a `tailf:snmp-oid` statement must be specified in the YANG header.

`--tagpath Path` — Generate the MIB only for a subtree of the module. The *Path* argument is an absolute schema node identifier, and it must refer to container nodes only.

`--import` *`Module Name`* — Add an IMPORT statement which imports *Name* from the MIB *Module*.

`--top` *`Name`* — Let *Name* be the name of the top object.

`--module` *`Name`* — Let *Name* be the module name. If a `tailf:snmp-mib-module-name` statement is in the YANG header, the two names must be equal.

`--generate-oids` — Translate all data nodes into MIB objects, and generate OIDs for data nodes without `tailf:snmp-oid` statements.

`--generate-yang-annotation` — Generate a YANG annotation file containing the `tailf:snmp-oid`, `tailf:snmp-mib-module-name` and `tailf:snmp-`

`row-status-column` statements for the nodes. Implies `--skip-symlinks`.

`--skip-symlinks` Do not generate MIB objects for data nodes modeled through symlinks.

`--fail-on-warnings` If this option is used all warnings are treated as errors and confdc will fail its execution.

`--no-comments` If this option is used no additional comments will be generated in the MIB.

`--read-only` If this option is used all objects in the MIB will be read only.

`--prefix` *`String`* Prefix all MIB object names with *String*.

Emit SNMP user options

`--snmpuser` *`EngineID User AuthType PrivType PassPhrase`* Generates a user entry with localized keys for the specified engine identifier. The output is an usmUserEntry in XML format that can be used in an initiation file for the SNMP-USER-BASED-SM-MIB::usmUserTable. In short this command provides key generation for users in SNMP v3. This option takes five arguments: The EngineID is either a string or a colon separated hexlist, or a dot separated octet list. The User argument is a string specifying the user name. The AuthType argument is one of md5, sha, or none. The PrivType argument is one of des, aes, or none. The PassPhrase argument is a string.

Emit W3C XML Schema options

`--emit-xsd` *`XsdFile`* Generate a W3C XML Schema file (.xsd file) from a .fxs file. This schema can be used to describe, using a standard notation, what actually goes over the wire.

`--xsd-import` Add xs:import constructs to generated W3C XML Schemas.

Emit Java options

`--emit-java` *`JFile`* Generate a .java ConfNamespace file from a .fxs file to be used when working with the Java library. The file is useful, but not necessary when working with the NAVU library. JFile could either be a file or a directory. If JFile is a directory the resulting .java file will be created in that directory with a name based on the module name in the YANG module. If JFile is not a directory that file is created. Use *--print-java-filename* to get the resulting file name.

`--print-java-filename` Only print the resulting java file name. Due to restrictions of identifiers in Java the name of the Class and thus the name of the file might get changed if non Java characters are used in the name of the file or in the name of the module. If this option is used no file is emitted the name of the file which would be created is just printed on stdout.

`--java-package` *`Package`* If this option is used the generated java file will have the given package declaration at the top.

`--exclude-enums` If this option is used, definitions for enums are omitted from the generated java file. This can in some cases be useful to avoid conflicts between enum symbols, or between enums and other symbols.

`--fail-on-warnings` If this option is used all warnings are treated as errors and confdc will fail its execution.

`-f`, `--fxsdep` *`FxsFile-OrDir`*... .fxs files (or directories containing .fxs files) to be used to resolve cross namespace dependencies.

Emit ConfM Java options

`--confm-emit-java` *`OutputDir`* Used to generate java classes for the ConfM Java library. The output is a number of data model aware java classes that are put in the specified OutputDir. These classes can be used with the ConfM library. For more information see the ConfM User Guide.

`--java-package` *`Package`* If this option is used the generated java file will have the given package declaration at the top.

We must also use this flag when we have multiple namespaces and one namespace refers to another. We can use the flag together with the *-c* flag, when we compile a YANG file.

`--java-class-naming` *`Level`* If this option is used the names of generated ConfM java classes will be constructed by prefixing the parent names of a node in order to make it's class name unique within the package. The level (0-N) argument controls how many parents that can be prefixed to the class name to make the name unique. Level 0 (which is the default) means that no parent names will be prefixed in order to attempt to make the name unique. The compiler will fail if there are any class naming conflicts. This is a way for the user to control how the java class names are constructed. Another way is to use annotations in the YANG module to control which java class name a specific tag will get.

`--java-class-naming-fixed` If this option is used the `--java-class-naming` flag will always prepend the parents to the generated name. If this option is not given it will only prepend parent names in order to make the name unique.

`--case-insensitive-naming` If this option is used the `--java-class-naming` option will not generate class names which clash on case insensitive file systems, e.g.. FAT32. For example, class names such as FooBar.java and FOOBar.java will not be created.

Check options

`--check-deps` Perform cross namespace dependency checking.

`-x`, `--xmlcheck` Verify that XmlFile contains valid XML according to FxsFile.

Misc options

`--get-info` *`FxsFile`* Various info about the file is printed on standard output, including the names of the source files used to produce this file, which confdc

version was used, and for fxs files, namespace URI, other namespaces the file depends on, namespace prefix, and mount point.

`--get-uri` *`FxsFile`* Extract the namespace URI.

`--version` Reports the confdc version.

EXAMPLE

Assume we have the file `system.yang`:

```
module system {
  namespace "http://example.com/ns/gargleblaster";
  prefix "gb";

  import ietf-inet-types {
    prefix inet;
  }
  container servers {
    list server {
      key name;
      leaf name {
        type string;
      }
      leaf ip {
        type inet:ip-address;
      }
      leaf port {
        type inet:port-number;
      }
    }
  }
}
```

To compile this file we do:

```
$ confdc -c system.yang -f ${CONFD_DIR}/src/confd/yang
```

We must provide a pointer to a directory that contains a compiled .fxs file for the ietf-inet-types module since we are importing that module.

If we intend to access data from this module from our C programs, it is meaningful to generate .h files like this:

```
$ confdc -c --emit-h blaster.h system.yang
```

The .h file contains `#define` entries for the different XML elements in system.yang. All C code that needs to manipulate or read data from this module must include the generated .h file.

```
confdc --emit-java blaster.java system.fxs
```

Finally we show how to compile a clispec into a loadable format:

```
$ confdc -c mycli.cli
$ ls mycli.ccl
myccl.ccl
```

DIAGNOSTICS

On success exit status is 0. On failure 1. Any error message is printed to stderr.

SEE ALSO

The ConfD User Guide

confd (1)	command to start and control the ConfD daemon
confd .conf(5)	ConfD daemon configuration file format
clispec(5)	CLI specification file format
mib_annotations(5)	MIB annotations file format

Name

maapi — command to access an ongoing transaction

Synopsis

`maapi` --get *`Path`*...

`maapi` --set *`Path Value`* [*`Path Value`*...]

`maapi` --keys *`Path`*...

`maapi` --exists *`Path`*...

`maapi` --delete *`Path`*...

`maapi` --create *`Path`*...

`maapi` --insert *`Path`*...

`maapi` --revert

`maapi` --msg *`To Message Sender`*
--priomsg *`To Message`*
--sysmsg *`To Message`*

`maapi` --cliget *`Param`*...

`maapi` --cliset *`Param Value`* [*`Param Value`*...]

`maapi` --cmd2path *`Cmd`* [*`Cmd`*]

`maapi` --cmd-path [--is-deleta] [--emit-parents] [--non-recursive] *`Path`* [*`Path`*]

`maapi` --cmd-diff *`Path`* [*`Path`*]

`maapi` --keypath-diff *`Path`*

`maapi` --clicmd [--get-io] [--no-hidden] [--no-error] [--no-aaa] [--no-fullpath] [--unhide <*`group`*>] *`Cli command`*...

DESCRIPTION

This command is intended to be used from inside a CLI command or a NETCONF extension RPC. These can be implemented in several ways, as an action callback or as an executable.

It is sometimes convenient to use a shell script to implement a CLI command and then invoke the script as an executable from the CLI. The maapi program makes it possible to manipulate the transaction in which the script was invoked.

Using the maapi command it is possible to, for example, write configuration wizards and custom show commands.

OPTIONS

`-g`, `--get` *`Path`* ... Read element value at Path and display result. Multiple values can be read by giving more than one Path as argument to get.

`-s, --set` *`Path Value ...`*	Set the value of Path to Value. Multiple values can be set by giving multiple Path Value pairs as arguments to set.
`-k, --keys` *`Path ...`*	Display all instances found at path. Multiple Paths can be specified.
`-e, --exists` *`Path ...`*	Exit with exit code 0 if Path exists (if multiple paths are given all must exist for the exit code to be 0).
`-d, --delete` *`Path ...`*	Delete element found at Path.
`-c, --create` *`Path ...`*	Create the element Path.
`-i, --insert` *`Path ...`*	Insert the element at Path. This is only possible if the elem has the 'indexed-view' attribute set.
`-z, --revert`	Remove all changes in the transaction.
`-m, --msg` *`To Message Sender`*	Send message to a user logged on to the system.
`-Q, --priomsg` *`To Message`*	Send prio message to a user logged on to the system.
`-M, --sysmsg` *`To Message`*	Send system message to a user logged on to the system.
`-G, --cliget` *`Param ...`*	Read and display CLI session parameter or attribute. Multiple params can be read by giving more than one Param as argument to cliget. Possible params are for C and I-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, output-file, screen-length, screen-width, history, terminal, autowizard, "service prompt config" , show-defaults, and if enabled, display-level. And for J-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, "output file", "screen length", "screen width", terminal, history, autowizard, "show defaults", and if enabled, display-level. In addition to this the attributes called annotation, tags and inactive can be read.
`-S, --cliset` *`Param Value ...`*	Set CLI session parameter to Value. Multiple params can be set by giving more than one Param-Value pair as argument to cliset. Possible params are for C and I-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, output-file, screen-length, screen-width, history, terminal, autowizard, "service prompt config", show-defaults, and if enabled, display-level. And for J-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, "output file", "screen length", "screen width", terminal, history, autowizard, "show defaults", and if enabled, display-level.
`-E, --cmd-path [--is-delete] [--emit-parents] [--non-recursive]` *`Path`*	Display the C- and I-style command for a given path. Optionally display the command to delete the path, and optionally emit the parents, ie the commands to reach the submode of the path.
`-L, --cmd-diff` *`Path`*	Display the C- and I-style command for going from the running configuration to the current configuration.
`-q, --keypath-diff` *`Path`*	Display the difference between the current state in the attached transaction and the running configuration. One line is emitted for each difference. Each such line begins with the type of the change,

	followed by a colon (':') character and lastly the keypath. The type of the change is one of the following: "created", "deleted", "modified", "value set", "moved after" and "attr set".
`-T`, `--cmd2path` *`Cmd`*	Attempts to derive an aaa-style namespace and path from a C-/I-style command path.
`-C`, `--clicmd`[`--get-io`][`--no-hidden`][`--no-error`][`--no-aaa`][`--no-fullpath`][`--unhide` *`group`*] *`Cli command to execute`*	Execute cli command in ongoing session, optionally ignoring that a command is hidden, unhiding a specific hide group, or ignoring the fullpath check of the argument to the show command. Multiple hide groups may be unhidden using the --unhide parameter multiple times.

EXAMPLE

Suppose we want to create an add-user wizard as a shell script. We would add the command in the clispec file `confd.cli` as follows:

```
...
 <configureMode>
   <cmd name="wizard">
     <info>Configuration wizards</info>
     <help>Configuration wizards</help>
     <cmd name="adduser">
       <info>Create a user</info>
       <help>Create a user</help>
       <callback>
         <exec>
           <osCommand>./adduser.sh</osCommand>
         </exec>
       </callback>
     </cmd>
   </cmd>
 </configureMode>
...
```

And have the following script `adduser.sh`:

```
#!/bin/bash

## Ask for user name
while true; do
    echo -n "Enter user name: "
    read user

    if [ ! -n "${user}" ]; then
 echo "You failed to supply a user name."
    elif maapi --exists "/aaa:aaa/authentication/users/user{${user}}"; then
 echo "The user already exists."
    else
 break
    fi
done

## Ask for password
while true; do
    echo -n "Enter password: "
    read -s pass1
```

```
    echo

    if [ "${pass1:0:1}" == "$" ]; then
 echo -n "The password must not start with $. Please choose a "
 echo    "different password."
    else
 echo -n "Confirm password: "
 read -s pass2
 echo

 if [ "${pass1}" != "${pass2}" ]; then
     echo "Passwords do not match."
 else
     break
 fi
    fi
done

groups=`maapi --keys "/aaa:aaa/authentication/groups/group"`
while true; do
    echo "Choose a group for the user."
    echo -n "Available groups are: "
    for i in ${groups}; do echo -n "${i} "; done
    echo
    echo -n "Enter group for user: "
    read group

    if [ ! -n "${group}" ]; then
 echo "You must enter a valid group."
    else
 for i in ${groups}; do
     if [ "${i}" == "${group}" ]; then
  # valid group found
  break 2;
     fi
 done
 echo "You entered an invalid group."
    fi
    echo
done

echo
echo "Creating user"
echo
maapi --create "/aaa:aaa/authentication/users/user{${user}}"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/password" \
 "${pass1}"

echo "Setting home directory to: /var/confd/homes/${user}"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/homedir" \
            "/var/confd/homes/${user}"
echo

echo "Setting ssh key directory to: "
echo "/var/confd/homes/${user}/ssh_keydir"
maapi --set "/aaa:aaa/authentication/users/user{${user}}/ssh_keydir" \
            "/var/confd/homes/${user}/ssh_keydir"
echo

maapi --set "/aaa:aaa/authentication/users/user{${user}}/uid" "1000"
```

```
maapi --set "/aaa:aaa/authentication/users/user{${user}}/gid" "100"

echo "Adding user to the ${group} group."
gusers=`maapi --get "/aaa:aaa/authentication/groups/group{${group}}/users"`

for i in ${gusers}; do
    if [ "${i}" == "${user}" ]; then
 echo "User already in group"
 exit 0
    fi
done
maapi --set "/aaa:aaa/authentication/groups/group{${group}}/users" \
           "${gusers} ${user}"
echo
exit 0
```

DIAGNOSTICS

On success exit status is 0. On failure 1 or 2. Any error message is printed to stderr.

ENVIRONMENT VARIABLES

Environment variables are used for determining which user session and transaction should be used when performing the operations. The CONFD_MAAPI_USID and CONFD_MAAPI_THANDLE environment variables are automatically set by ConfD when invoking a CLI command, but when a NETCONF extension RPC is invoked, only CONFD_MAAPI_USID is set, since there is no transaction associated with such an invocation.

`CONFD_MAAPI_USID`	User session to use.
`CONFD_MAAPI_THANDLE`	The transaction to use when performing the operations.
`CONFD_MAAPI_DEBUG`	Maapi debug information will be printed if this variable is defined.
`CONFD_IPC_ADDR`, `CONFD_IPC_EXTADDR`	The address used to connect to the ConfD daemon, overrides the compiled in default.
`CONFD_IPC_PORT`	The port number to connect to the ConfD daemon on, overrides the compiled in default.
`CONFD_IPC_EXTSOPATH`	The absolute path to the shared object to use for a connection using external IPC when `CONFD_IPC_EXTADDR` is given.

SEE ALSO

The ConfD User Guide

confd(1) - command to start and control the ConfD daemon

confdc(1) - YANG compiler

confd.conf(5) - ConfD daemon configuration file format

clispec(5) - CLI specification file format

Name

pyang — validate and convert YANG modules to various formats

Synopsis

`pyang` [--verbose] [--canonical] [--strict] [--ietf] [--lax-xpath-checks] [--hello] [--check-update-from *`oldfile`*] [-o *`outfile`*] [-f *`format`*] [-p *`path`*] [-W *`warning`*] [-E *`error`*] [-a *`filename`*] [[--no-features] | [-F *`feature`*]] [--ignore-unknown-features] *`file`* ...

`pyang` --tailf-sanitize [--tailf-remove-body] [--tailf-keep-actions] [--tailf-keep-info] [--tailf-keep-tailf-typedefs] [--tailf-keep-symlink-when] [--tailf-keep-symlink-must] [--tailf-keep-display-when]

`pyang` -h | --help

`pyang` --rest-doc [--rest-doc-db] [--rest-doc-address] [--rest-doc-output] [--rest-doc-query-type] [--rest-doc-end-list-url-with-keys] [--rest-doc-end-list-url-with-no-keys] [--rest-doc-list-keys] [--rest-doc-path] [--rest-doc-curl-flags]

`pyang` -v | --version

One or more *`file`* parameters may be given on the command line. They denote either YANG modules to be processed (in YANG or YIN syntax) or, using the `--hello` switch, a server <hello> message conforming to RFC 6241 and RFC 6020, which completely defines the data model - supported YANG modules as well as features and capabilities. In the latter case, only one *`file`* parameter may be present.

If no files are given, **pyang** reads input from stdin, which must be one module or a server <hello> message.

Description

The **pyang** program is used to validate YANG modules (RFC 6020). It is also used to convert YANG modules into equivalent YIN modules. From a valid module a hybrid DSDL schema (RFC 6110) can be generated.

If no *`format`* is given, the specified modules are validated, and the program exits with exit code 0 if all modules are valid.

Options

`-h --help`	Print a short help text and exit.
`-v --version`	Print the version number and exit.
`-e --list-errors`	Print a listing of all error codes and messages pyang might generate, and then exit.
`--print-error-code`	On errors, print the symbolic error code instead of the error message.
`-Werror`	Treat warnings as errors.
`-Wnone`	Do not print any warnings.
`-W` *`errorcode`*	Treat *`errorcode`* as a warning, even if `-Werror` is given. *`errorcode`* must be a warning or a minor error. Use `--list-errors` to get a listing of all errors and warnings.

The following example treats all warnings except the warning for unused imports as errors:

```
$ pyang --Werror -W UNUSED_IMPORT
```

`-E` *`errorcode`*
: Treat the warning *`errorcode`* as an error.

  Use `--list-errors` to get a listing of all errors and warnings.

  The following example treats only the warning for unused import as an error:

```
$ pyang --Werror -W UNUSED_IMPORT
```

`--ignore-errors`
: Ignore errors. Use with care. Plugins that don't expect to be invoked if there are errors present may crash.

`--keep-comments`
: This parameter has effect only if a plugin can handle comments. One example of such a plugin is the 'yang' output format plugin.

`--canonical`
: Validate the module(s) according to the canonical YANG order.

`--strict`
: Force strict YANG compliance. Currently this checks that the deref() function is not used in XPath expressions and leafrefs.

`--ietf`
: Validate the module(s) according to IETF rules as specified in RFC 6087. In addition, it checks that the module is in canonical order, and that `--max-line-length` is 72 so that the module fits into an RFC.

`--lax-xpath-checks`
: Lax checks of XPath expressions. Specifically, do not generate an error if an XPath expression uses a variable or an unknown function.

`-L --hello`
: Interpret the input file or standard input as a server <hello> message. In this case, no more than one *`file`* parameter may be given.

`--trim-yin`
: In YIN input modules, remove leading and trailing whitespace from every line in the arguments of the following statements: 'contact', 'description', 'error-message', 'organization' and 'reference'. This way, the XML-indented argument texts look tidy after translating the module to the compact YANG syntax.

`--max-line-length` *`maxlen`*
: Give a warning if any line is longer than *`maxlen`*.

`--max-identifier-length` *`maxlen`*
: Give a error if any identifier is longer than *`maxlen`*.

`-f --format` *`format`*
: Convert the module(s) into *`format`*. Some translators require a single module, and some can translate multiple modules at one time. If no *`outfile`* file is specified, the result is printed on stdout. The supported formats are listed in Output Formats below.

`-o --output` *`outfile`*
: Write the output to the file *`outfile`* instead of stdout.

`--features` *`features`*
: *`features`* is a string on the form *`modulename`*:[*`feature`*(,*`feature`*)*]

This option is used to prune the data model by removing all nodes that are defined with a "if-feature" that is not listed as *feature*. This option affects all output formats.

This option can be given multiple times, and may be also combined with `--hello`. If a `--features` option specifies different supported features for a module than the hello file, the `--features` option takes precedence.

If this option is not given, nothing is pruned, i.e., it works as if all features were explicitly listed.

For example, to view the tree output for a module with all if-feature'd nodes removed, do:

```
$ pyang -f tree --features mymod: mymod.yang
```

`--deviation-module` *file*

This option is used to apply the deviations defined in *file*. This option affects all output formats.

This option can be given multiple times.

For example, to view the tree output for a module with some deviations applied, do:

```
$ pyang -f tree --deviation-module mymod-devs.yang mymod.yang
```

`-p --path` *path*

path is a colon (:) separated list of directories to search for imported modules. This option may be given multiple times.

The following directories are always added to the search path:

1. current directory
2. `$YANG_MODPATH`
3. `$HOME/yang/modules`
4. `$YANG_INSTALL/yang/modules` OR if `$YANG_INSTALL` is unset `<the default installation directory>/yang/modules` (on Unix systems: `/usr/share/yang/modules`)

`--plugindir` *plugindir*

Load all YANG plugins found in the directory *plugindir*. This option may be given multiple times.

list of directories to search for pyang plugins. The following directories are always added to the search path:

1. `pyang/plugins` from where pyang is installed

2. `$PYANG_PLUGINPATH`

`--check-update-from` *oldfile*

Checks that a new revision of a module follows the update rules given in RFC 6020. *oldfile* is the old module and *file* is the new version of the module.

If the old module imports or includes any modules or submodules, it is important that the the old versions of these modules and submodules are found. By default, the directory where *oldfile* is found

is used as the only directory in the search path for old modules. Use the option `--check-update-from-path` to control this path.

`-P --check-update-from-path` *`oldpath`*
: *`oldpath`* is a colon (:) separated list of directories to search for imported modules. This option may be given multiple times.

`file...`
: These are the names of the files containing the modules to be validated, or the module to be converted.

Tail-f Specific Options

`-a --annotate` *`filename`*
: *filename* is the name of a file containing a YANG module with tailf:annotate statements.

  This parameter can be given multiple times.

`--cs-feature` *`feature`*
: Indicates that support for *feature* should be present in the generated confspec file. If *feature* is defined in an imported YANG module, it must be given as *prefix:feature-name*.

  If no such parameter is given, all features in the module is supported.

  This parameter can be given multiple times.

`--cs-no-features`
: Indicates that no features are supported.

`--cs-ignore-unknown-features`
: Instructs the compiler to not give an error if an unknown feature is specified with `--feature`.

`--cs-use-description`
: Normally, 'description' statements are ignored by pyang when translating to confspecs. Instead the 'tailf:info' statement is used as help and information text in the CLI and WebUI. When this option is specified, text in 'description' statements is used if no 'tailf:info' statement is present.

Tail-f Sanitation Options

`--tailf-sanitize`
: This option removes all tailf specific statements from a module, including the import statement that imports tailf-common, if possible. This option is intended to be used together with `-f yang` or `-f yin`, in order to produce a module without any references to tailf modules.

  This option is useful in order to publish YANG modules to NETCONF clients, like Tail-f's NCS.

  By default, this option:

  - Removes all tailf extension statements. Most of these extensions are used to control the implementation on the server, or to tweak the CLI experience. These extensions are not needed in a NETCONF client.
  - Removes any 'must' or 'when' statement that uses a non-standard XPath function.

- Removes any node marked with a 'tailf:hidden full' statement. Such a node is not visible in any agent.
- Inlines the subtree that a 'tailf:symlink' points to. If any node in the target subtree contains 'when' or 'must' statements, these are removed. The reason for this is that in general, it is not possible to guarantee that the argument to such a statement will point to a node that is visible in the module.
- Copies any typedefs from 'tailf-common' into the module, and changes any references to such types to reference the copied type.

`--tailf-remove-body` Removes all 'container', 'list', 'leaf', 'leaf-list', 'augment', 'rpc', and 'notification' statements from the module.

This can be useful if the module is normally compiled with the `--export` option to **confdc**, and the module is not exported to NETCONF, but it still has typedefs, groupings, features, or identities that are referenced by some other module. By using this option, such a module can safely be published to a NETCONF client.

`--tailf-keep-non-std-xpath` In the sanitition process, do not remove 'must' and 'when' statements with non-standard XPath functions.

If a module is prepared for NCS, this option should be used, so that NCS can make use of the 'must' and 'when' statements.

`--tailf-keep-actions` In the sanitition process, do not remove 'tailf:action' statements. This is useful if the client understands this extension.

If a module is prepared for NCS, this option should be used, so that NCS can invoke actions on the server.

`--tailf-keep-info` In the sanitition process, do not remove 'tailf:info' statements. This is useful if the client understands this extension.

If a module is prepared for NCS, this option should be used, so that the NCS CLI prints the given help text.

`--tailf-keep-tailf-type-defs` In the sanitation process, do not copy types from 'tailf-common'. This option can be given if the 'tailf-common' module is published to the client.

If a module is prepared for NCS, this option should be used, so that the NCS uses the correct internal representation of these types.

`--tailf-keep-sym-link-when` In the sanitation process, do not remove 'when' statements found in the 'tailf:symlink' target subtree. Use this option only if the 'when' expressions do not escape out of the symlinked subtree.

`--tailf-keep-sym-link-must` In the sanitation process, do not remove 'must' statements found in the 'tailf:symlink' target subtree. Use this option only if the 'must' expressions do not escape out of the symlinked subtree.

`--tailf-keep-dis-play-when` In the sanitition process, do not remove 'tailf:display-when' statements. This is useful if the client understands this extension.

Confspec Generation

In order to compile a YANG module for usage with ConfD, the YANG (or YIN) module is converted to a functionally equivalent confspec. This confspec is then compiled by the normal ConfD toolchain. For example:

```
$ pyang -f cs test.yang -o test.cs
$ confdc -c test.cs
$ confdc -l -o t.fxs t.xso
```

REST Documentation Options

`--rest-doc-db`=[candidate | running]

This option defines what database to use, default is *running*.

`--rest-doc-address`=*address*

Address used in curl output, default is *http://localhost:8080*.

`--rest-doc-output`=[delete | delete_curl | docbook | get | get_curl | patch | patch_curl | post | post_curl | put | put_curl]

This option defines the output format, default is *get*.

- *delete* output generic DELETE format.
- *delete_curl* output curl DELETE format.
- *docbook* output docbook format.

  The docbook format contains delete, delete_curl, get, get_curl, patch, patch_curl, put, put_curl, post and post_curl formats for all the top containers and lists in a module.
- *get* output generic GET format.
- *get_curl* output curl GET format.
- *patch* generates the generic PATCH format.
- *patch_curl* generates the curl PATCH format.
- *post* generates the generic POST format.
- *post_curl* generates the curl POST format.
- *put* generates the generic PUT format.
- *put_curl* generates the curl PUT format.

`--rest-doc-query-type`=[default | shallow | deep]

GET result depth, default is *default*.

`--rest-doc-end-list-url-with-keys`

End example list path with key parameters. This is the default.

`--rest-doc-end-list-url-with-no-keys`

Don't end example list path with key parameters.

`--rest-doc-list-keys`

JSON dictionary of list keys and values.

Replaces the example leaf- and key-values in GET, PATCH, POST and PUT, with the specified values in the dictionary.

`--rest-doc-path` Keypath to the yang-statment to generate output for.

`--rest-doc-curl-flags` Additional curl-flags added to the --rest-doc-output *_curl commands.

OUTPUT EXAMPLES

--rest-doc-output=get

```
# leaf (string) /t1:test/t1:leaf-j-1/t1:s1
GET /api/running/test/leaf-j-1/s1
Accept: application/vnd.yang.data+json
```

--rest-doc-output=get_curl

```
curl -X GET \
  -u admin:admin \
  -H "Accept: application/vnd.yang.data+json" \
  http://localhost:8080/api/running/test/leaf-j-1/s1
```

--rest-doc-output=patch_curl

```
echo '
{
  "cont-j-1" : {
    "s1" : "s1 value",
    "i1" : 42,
    "ui1" : 42
  }
}
' | curl -X PATCH -d @- -u admin:admin \
      -H "Content-type: application/vnd.yang.data+json" \
      http://localhost:8080/api/running/test/cont-j-1
```

Output Formats

Installed **pyang** plugins may define their own options, or add new formats to the `-f` option. These options and formats are listed in **`pyang -h`**.

capability Capability URIs for each module of the data model.

depend Makefile dependency rule for the module.

dsdl Hybrid DSDL schema, see RFC 6110.

hypertree Hyperbolic tree navigator that can be displayed by **treebolic**.

jsonxsl XSLT stylesheet for transforming XML instance documents to JSON.

jstree	HTML/JavaScript tree navigator.
jtox	Driver file for transforming JSON instance documents to XML.
omni	An applescript file that draws a diagram in **OmniGraffle**.
sample-xml-skeleton	Skeleton of a sample XML instance document.
tree	Tree structure of the module.
uml	UML file that can be read by **plantuml** to generate UML diagrams.
xmi	XMI file that can be imported by **ArgoUML**.
xsd	DEPRECATED: W3C XML Schema.
yang	Normal YANG syntax.
yin	The XML syntax of YANG.

<varlistentry>*cs*
Tail-f confspec language
</varlistentry>

Capability Output

The *capability* prints a capability URL for each module of the input data model, taking into account features and deviations, as described in section 5.6.4 of RFC 6020.

Options for the *capability* output format:

`--capability-entity`	Write ampersands in the output as XML entities ("&").

Depend Output

The *depend* output generates a Makefile dependency rule for files based on a YANG module. This is useful if files are generated from the module. For example, suppose a .c file is generated from each YANG module. If the YANG module imports other modules, or includes submodules, the .c file needs to be regenerated if any of the imported or included modules change. Such a dependency rule can be generated like this:

```
$ pyang -f depend --depend-target mymod.c \
      --depend-extension .yang mymod.yang
      mymod.c : ietf-yang-types.yang my-types.yang
```

Options for the *depend* output format:

`--depend-target`	Makefile rule target. Default is the module name.
`--depend-extension`	YANG module file name extension. Default is no extension.
`--depend-no-submodules`	Do not generate dependencies for included submodules.
`--depend-include-path`	Include file path in the prerequisites. Note that if no `--depend-extension` has been given, the prerequisite is the filename as found, i.e., ending in ".yang" or ".yin".

`--depend-ignore-module` Name of YANG module or submodule to ignore in the prerequisites. This option can be given multiple times.

DSDL Output

The *dsdl* output takes a data model consisting of one or more YANG modules and generates a hybrid DSDL schema as described in RFC 6110. The hybrid schema is primarily intended as an interim product used by **yang2dsdl**(1).

The *dsdl* plugin also supports metadata annotations, if they are defined and used as described in draft-lhotka-netmod-yang-metadata.

Options for the *dsdl* output format:

`--dsdl-no-documentation` Do not print documentation annotations

`--dsdl-no-dublin-core` Do not print Dublin Core metadata terms

`--dsdl-record-defs` Record translations of all top-level typedefs and groupings in the output schema, even if they are not used. This is useful for translating library modules.

hypertree output

The *hypertree* output generates a hyperbolic YANG browser. The generated xml file can be imported to **treebolic** (`http://treebolic.sourceforge.net/en/`).

Color coding in the tree:

- Light green node background : config = True
- Light yellow node background : config = False
- Red node foreground : mandatory = True
- White leaf node background : index
- Orange foreground : presence container

The xml file references an images folder that needs to exist in the same folder as the generated file. This is installed as share/yang/images in the pyang installation directory. The easiest way is to symlink to this directory.

`pyang -f hypertree model.yang -o model.xml`

Prepare a HTML file that links to the generated XMI file:

```
<applet code="treebolic.applet.Treebolic.class"
archive="TreebolicAppletDom.jar"
id="Treebolic" width="100%" height="100%">
<param name="doc" value="model.xml">
</applet>
```

hypertree output specific option:

`--hypertree-help` Print help on hypertree usage and exit.

`--xmi-no-assoc-names` Do not print association names. ArgoUML has no way of hiding the association name and the diagram gets cluttered.

JSONXSL Output

The *jsonxsl* output generates an XSLT 1.0 stylesheet that can be used for transforming an XML instance document into JSON text as specified in draft-ietf-netmod-yang-json. The XML document must be a valid instance of the data model which is specified as one or more input YANG modules on the command line (or via a <hello> message, see the `--hello` option).

The data tree(s) must be wrapped at least in either <nc:data> or <nc:config> element, where "nc" is the namespace prefix for the standard NETCONF URI "urn:ietf:params:xml:ns:netconf:base:1.0", or the XML instance document has to be a complete NETCONF RPC request/reply or notification. Translation of RPCs and notifications defined by the data model is also supported.

The generated stylesheet accepts the following parameters that modify its behaviour:

- *compact*: setting this parameter to 1 results in a compact representation of the JSON text, i.e. without any whitespace. The default is 0 which means that the JSON output is pretty-printed.
- *ind-step*: indentation step, i.e. the number of spaces to use for each level. The default value is 2 spaces. Note that this setting is only useful for pretty-printed output (compact=0).

The stylesheet also includes the file `jsonxsl-templates.xsl` which is a part of **pyang** distribution.

jstree Output

The *jstree* output grenerates an HTML/JavaScript page that presents a tree-navigator to the given YANG module(s).

jstree output specific option:

`--jstree-no-path` Do not include paths in the output. This option makes the page less wide.

JTOX Output

The *jtox* output generates a driver file which can be used as one of the inputs to **json2xml** for transforming a JSON document to XML as specified in draft-ietf-netmod-yang-json.

The *jtox* output itself is a JSON document containing a concise representation of the data model which is specified as one or more input YANG modules on the command line (or via a <hello> message, see the `--hello` option).

See **json2xml** manual page for more information.

Omni Output

The plugin generates an applescript file that draws a diagram in OmniGraffle. Requires OmniGraffle 6. Usage:

```
$ pyang -f omni foo.yang -o foo.scpt
$ osascript foo.scpt
```

omni output specific option:

`--omni-path` *`path`* — Subtree to print. The *`path`* is a slash ("/") separated path to a subtree to print. For example "/nacm/groups".

Sample-xml-skeleton Output

The *sample-xml-skeleton* output generates an XML instance document with sample elements for all nodes in the data model, according to the following rules:

- An element is present for every leaf, container or anyxml.
- At least one element is present for every leaf-list or list. The number of entries in the sample is min(1, min-elements).
- For a choice node, sample element(s) are present for each case.
- Leaf, leaf-list and anyxml elements are empty (but see the `--sample-xml-skeleton-defaults` option below).

Note that the output document will most likely be invalid and needs manual editing.

Options specific to the *sample-xml-skeleton* output format:

`--sample-xml-skeleton-doctype=`*`type`* — Type of the sample XML document. Supported values for *`type`* are `data` (default) and `config`. This option determines the document element of the output XML document (<data> or <config> in the NETCONF namespace) and also affects the contents: for `config`, only data nodes representing configuration are included.

`--sample-xml-skeleton-defaults` — Add leaf elements with defined defaults to the output with their default value. Without this option, the default elements are omitted.

`--sample-xml-skeleton-annotations` — Add XML comments to the sample documents with hints about expected contents, for example types of leaf nodes, permitted number of list entries etc.

Tree Output

The *tree* output prints the resulting schema tree from one or more modules. Use **`pyang --tree-help`** to print a description on the symbols used by this format.

Tree output specific options:

`--tree-help` — Print help on symbols used in the tree output and exit.

`--tree-depth` *`depth`* — Levels of the tree to print.

`--tree-path` *`path`* — Subtree to print. The *`path`* is a slash ("/") separated path to a subtree to print. For example "/nacm/groups".

UML Output

The *uml* output prints an output that can be used as input-file to **plantuml** (`http://plantuml.sourceforge.net`) in order to generate a UML diagram. Note that it requires **graphviz** (`http://www.graphviz.org/`).

For large diagrams you may need to increase the Java heap-size by the -XmxSIZEm option, to java. For example: **`java -Xmx1024m -jar plantuml.jar ....`**

Options for the *UML* output format:

`--uml-classes-only`	Generate UML with classes only, no attributes
`--uml-split-pages=`*`layout`*	Generate UML output split into pages, NxN, example 2x2. One .png file per page will be rendered.
`--uml-out-put-directory=`*`directory`*	Put the generated .png files(s) in the specified output directory. Default is "img/"
`--uml-title=`*`title`*	Set the title of the generated UML diagram, (default is YANG module name).
`--uml-header=`*`header`*	Set the header of the generated UML diagram.
`--uml-footer=`*`footer`*	Set the footer of the generated UML diagram.
`--uml-long-identifers`	Use complete YANG schema identifiers for UML class names.
`--uml-no=`*`arglist`*	This option suppresses specified arguments in the generated UML diagram. Valid arguments are: uses, leafref, identity, identityref, typedef, annotation, import, circles, stereotypes. Annotation suppresses YANG constructs rendered as annotations, examples module info, config statements for containers. Example --uml-no=circles,stereotypes,typedef,import
`--uml-truncate=`*`elemlist`*	Leafref attributes and augment elements can have long paths making the classes too wide. This option will only show the tail of the path. Example --uml-truncate=augment,leafref.
`--uml-inline-groupings`	Render the diagram with groupings inlined.
`--uml-inline-augments`	Render the diagram with augments inlined.
`--uml-max-enums=`*`number`*	Maximum of enum items rendered.
`--uml-filter-file=`*`file`*	NOT IMPLEMENTED: Only paths in the filter file will be included in the diagram. A default filter file is generated by option --filter.

XMI Output

The *xmi* output prints an XMI file that can be imported by ArgUML http://argouml.tigris.org/.

Drag all classes to the diagram area in ArgoUML and use the Arrange-Layout menu.

XMI output specific option:

XSD Output

NOTE: The XSD output plugin is deprecated. Use the dsdl plugin instead.

Options for the *xsd* output format:

`--xsd-no-appinfo`	Do not print YANG specific appinfo.

`--xsd-no-lecture`	Do not print the lecture about how the XSD can be used.
`--xsd-no-imports`	Do not generate any xs:imports.
`--xsd-break-pattern`	Break long patterns so that they fit into RFCs. The resulting patterns might not always be valid XSD, so use with care.

YANG Output

Options for the *yang* output format:

`--yang-canonical`	Generate all statements in the canonical order.
`--yang-remove-unused-imports`	Remove unused import statements from the output.

YIN Output

Options for the *yin* output format:

`--yin-canonical`	Generate all statements in the canonical order.
`--yin-pretty-strings`	Pretty print strings, i.e. print with extra whitespace in the string. This is not strictly correct, since the whitespace is significant within the strings in XML, but the output is more readable.

YANG Extensions

This section describes XPath functions that can be used in "must", "when", or "path" expressions in YANG modules, in addition to the core XPath 1.0 functions.

pyang can be instructed to reject the usage of these functions with the parameter `--strict`.

Function: *node-set* **deref**(*node-set*)

The `deref` function follows the reference defined by the first node in document order in the argument node-set, and returns the nodes it refers to.

If the first argument node is an instance-identifier, the function returns a node-set that contains the single node that the instance identifier refers to, if it exists. If no such node exists, an empty node-set is returned.

If the first argument node is a leafref, the function returns a node-set that contains the nodes that the leafref refers to.

If the first argument node is of any other type, an empty node-set is returned.

The following example shows how a leafref can be written with and without the `deref` function:

```
/* without deref */

leaf my-ip {
  type leafref {
    path "/server/ip";
  }
}
```

```
leaf my-port {
  type leafref {
    path "/server[ip = current()/../my-ip]/port";
  }
}

/* with deref */

leaf my-ip {
  type leafref {
    path "/server/ip";
  }
}
leaf my-port {
  type leafref {
    path "deref(../my-ip)/../port";
  }
}
```

Example

The following example validates the standard YANG modules with derived types:

```
$ pyang ietf-yang-types.yang ietf-inet-types.yang
```

The following example converts the ietf-yang-types module into YIN:

```
$ pyang -f yin -o ietf-yang-types.yin ietf-yang-types.yang
```

The following example converts the ietf-netconf-monitoring module into a UML diagram:

```
       $ pyang -f uml ietf-netconf-monitoring.yang > \
       ietf-netconf-monitoring.uml
       $ java -jar plantuml.jar ietf-netconf-monitoring.uml
       $ open img/ietf-netconf-monitoring.png
```

Environment Variables

pyang searches for referred modules in the colon (:) separated path defined by the environment variable `$YANG_MODPATH` and in the directory `$YANG_INSTALL/yang/modules`.

pyang searches for plugins in the colon (:) separated path defined by the environment variable `$PYANG_PLUGINDIR`.

Bugs

1. The XPath arguments for the *must* and *when* statements are checked only for basic syntax errors.

ConfD man-pages, Volume 3

Table of Contents

Name

confd_lib — C library for connecting to ConfD

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to ConfD. The documentation for the library is divided into several manual pages:

confd_lib_lib(3)	Common Library Functions
confd_lib_dp(3)	The Data Provider API
confd_lib_events(3)	The Event Notification API
confd_lib_ha(3)	The High Availability API
confd_lib_cdb(3)	The CDB API
confd_lib_maapi(3)	The Management Agent API

There is also a C header file associated with each of these manual pages:

`#include <confd_lib.h>`	Common type definitions and prototypes for the functions in the confd_lib_lib(3) manual page. Always needed.
`#include <confd_dp.h>`	Needed when functions in the confd_lib_dp(3) manual page are used.
`#include <confd_events.h>`	Needed when functions in the confd_lib_events(3) manual page are used.
`#include <confd_ha.h>`	Needed when functions in the confd_lib_ha(3) manual page are used.
`#include <confd_cdb.h>`	Needed when functions in the confd_lib_cdb(3) manual page are used.
`#include <confd_maapi.h>`	Needed when functions in the confd_lib_maapi(3) manual page are used.

For backwards compatibility, `#include <confd.h>` can also be used, and is equivalent to:

```
#include <confd_lib.h>
#include <confd_dp.h>
#include <confd_events.h>
#include <confd_ha.h>
```

SEE ALSO

The ConfD User Guide

Name

confd_lib_cdb — library for connecting to ConfD built-in XML database (CDB)

Synopsis

#include <confd_lib.h> #include <confd_cdb.h>

```
int cdb_connect(int sock, enum cdb_sock_type type, const struct sock-
addr* srv, int srv_sz);

int cdb_connect_name(int sock, enum cdb_sock_type type, const struct
sockaddr* srv, int srv_sz, const char *name);

int cdb_mandatory_subscriber(int sock, const char *name);

int cdb_set_namespace(int sock, int hashed_ns);

int cdb_end_session(int sock);

int cdb_start_session(int sock, enum cdb_db_type db);

int cdb_start_session2(int sock, enum cdb_db_type db, int flags);

int cdb_close(int sock);

int cdb_wait_start(int sock);

int cdb_get_phase(int sock, struct cdb_phase *phase);

int cdb_get_txid(int sock, struct cdb_txid *txid);

int cdb_initiate_journal_compaction(int sock);

int cdb_load_file(int sock, const char *filename, int flags);

int cdb_load_str(int sock, const char *xml_str, int flags);

int cdb_get_user_session(int sock);

int cdb_get_transaction_handle(int sock);

int cdb_set_timeout(int sock, int timeout_secs);

int cdb_exists(int sock, const char *fmt, ...);

int cdb_cd(int sock, const char *fmt, ...);

int cdb_pushd(int sock, const char *fmt, ...);

int cdb_popd(int sock);

int cdb_getcwd(int sock, size_t strsz, char *curdir);

int cdb_getcwd_kpath(int sock, confd_hkeypath_t **kp);

int cdb_num_instances(int sock, const char *fmt, ...);
```

int **cdb_next_index**(int *sock*, const char **fmt*, ...);

int **cdb_index**(int *sock*, const char **fmt*, ...);

int **cdb_is_default**(int *sock*, const char **fmt*, ...);

int **cdb_subscribe2**(int *sock*, enum cdb_sub_type *type*, int *flags*, int *priority*, int **spoint*, int *nspace*, const char **fmt*, ...);

int **cdb_subscribe**(int *sock*, int *priority*, int *nspace*, int **spoint*, const char **fmt*, ...);

int **cdb_oper_subscribe**(int *sock*, int *nspace*, int **spoint*, const char **fmt*, ...);

int **cdb_subscribe_done**(int *sock*);

int **cdb_trigger_subscriptions**(int *sock*, int *sub_points[]*, int *len*);

int **cdb_trigger_oper_subscriptions**(int *sock*, int *sub_points[]*, int *len*, int *flags*);

int **cdb_diff_match**(int *sock*, int *subid*, struct xml_tag *tags[]*, int *tagslen*);

int **cdb_read_subscription_socket**(int *sock*, int *sub_points[]*, int **resultlen*);

int **cdb_read_subscription_socket2**(int *sock*, enum cdb_sub_notification **type*, int **flags*, int **subpoints[]*, int **resultlen*);

int **cdb_replay_subscriptions**(int *sock*, struct cdb_txid **txid*, int *sub_points[]*, int *len*);

int **cdb_get_replay_txids**(int *sock*, struct cdb_txid ***txid*, int **resultlen*);

int **cdb_diff_iterate**(int *sock*, int *subid*, enum cdb_iter_ret (**iter*)(confd_hkeypath_t **kp*, enum cdb_iter_op *op*, confd_value_t **oldv*, confd_value_t **newv*, void **state*), int *flags*, void **initstate*);

int **cdb_diff_iterate_resume**(int *sock*, enum cdb_iter_ret *reply*, enum cdb_iter_ret (**iter*)(confd_hkeypath_t **kp*, enum cdb_iter_op *op*, confd_value_t **oldv*, confd_value_t **newv*, void **state*), void **resumestate*);

int **cdb_cli_diff_iterate**(int *sock*, int *subid*, cli_diff_iter_function_t **iter*, int *flags*, void **initstate*);

int **cdb_get_modifications**(int *sock*, int *subid*, int *flags*, confd_tag_value_t ***values*, int **nvalues*, const char **fmt*, ...);

int **cdb_get_modifications_iter**(int *sock*, int *flags*, confd_tag_value_t ***values*, int **nvalues*);

int **cdb_get_modifications_cli**(int *sock*, int *subid*, int *flags*, char ***str*);

int **cdb_sync_subscription_socket**(int *sock*, enum cdb_subscription_sync_type *st*);

int **cdb_sub_progress**(int *sock*, const char *_fmt_, ...);

int **cdb_sub_abort_trans**(int *sock*, enum confd_errcode *code*, u_int32_t *apptag_ns*, u_int32_t *apptag_tag*, const char *_fmt_, ...);

int **cdb_sub_abort_trans_info**(int *sock*, enum confd_errcode *code*, u_int32_t *apptag_ns*, u_int32_t *apptag_tag*, const confd_tag_value_t *_error_info_, int *n*, const char *_fmt_, ...);

int **cdb_get_case**(int *sock*, const char *_choice_, confd_value_t *_rcase_, const char *_fmt_, ...);

int **cdb_get**(int *sock*, confd_value_t *_v_, const char *_fmt_, ...);

int **cdb_get_int8**(int *sock*, int8_t *_rval_, const char *_fmt_, ...);

int **cdb_get_int16**(int *sock*, int16_t *_rval_, const char *_fmt_, ...);

int **cdb_get_int32**(int *sock*, int32_t *_rval_, const char *_fmt_, ...);

int **cdb_get_int64**(int *sock*, int64_t *_rval_, const char *_fmt_, ...);

int **cdb_get_u_int8**(int *sock*, u_int8_t *_rval_, const char *_fmt_, ...);

int **cdb_get_u_int16**(int *sock*, u_int16_t *_rval_, const char *_fmt_, ...);

int **cdb_get_u_int32**(int *sock*, u_int32_t *_rval_, const char *_fmt_, ...);

int **cdb_get_u_int64**(int *sock*, u_int64_t *_rval_, const char *_fmt_, ...);

int **cdb_get_bit32**(int *sock*, u_int32_t *_rval_, const char *_fmt_, ...);

int **cdb_get_bit64**(int *sock*, u_int64_t *_rval_, const char *_fmt_, ...);

int **cdb_get_ipv4**(int *sock*, struct in_addr *_rval_, const char *_fmt_, ...);

int **cdb_get_ipv6**(int *sock*, struct in6_addr *_rval_, const char *_fmt_, ...);

int **cdb_get_double**(int *sock*, double *_rval_, const char *_fmt_, ...);

int **cdb_get_bool**(int *sock*, int *_rval_, const char *_fmt_, ...);

int **cdb_get_datetime**(int *sock*, struct confd_datetime *_rval_, const char *_fmt_, ...);

int **cdb_get_date**(int *sock*, struct confd_date *_rval_, const char *_fmt_, ...);

int **cdb_get_time**(int *sock*, struct confd_time *_rval_, const char *_fmt_, ...);

int **cdb_get_duration**(int *sock*, struct confd_duration *_rval_, const char *_fmt_, ...);

int **cdb_get_enum_value**(int *sock*, int32_t *_rval_, const char *_fmt_, ...);

int **cdb_get_objectref**(int *sock*, confd_hkeypath_t *******rval*, const char **fmt*, ...);

int **cdb_get_oid**(int *sock*, struct confd_snmp_oid *******rval*, const char **fmt*, ...);

int **cdb_get_buf**(int *sock*, unsigned char *******rval*, int **bufsiz*, const char **fmt*, ...);

int **cdb_get_buf2**(int *sock*, unsigned char **rval*, int **n*, const char **fmt*, ...);

int **cdb_get_str**(int *sock*, char **rval*, int *n*, const char **fmt*, ...);

int **cdb_get_binary**(int *sock*, unsigned char ***rval*, int **bufsiz*, const char **fmt*, ...);

int **cdb_get_hexstr**(int *sock*, unsigned char ***rval*, int **bufsiz*, const char **fmt*, ...);

int **cdb_get_qname**(int *sock*, unsigned char ***prefix*, int **prefixsz*, unsigned char ***name*, int **namesz*, const char **fmt*, ...);

int **cdb_get_list**(int *sock*, confd_value_t ***values*, int **n*, const char **fmt*, ...);

int **cdb_get_ipv4prefix**(int *sock*, struct confd_ipv4_prefix **rval*, const char **fmt*, ...);

int **cdb_get_ipv6prefix**(int *sock*, struct confd_ipv6_prefix **rval*, const char **fmt*, ...);

int **cdb_get_decimal64**(int *sock*, struct confd_decimal64 **rval*, const char **fmt*, ...);

int **cdb_get_identityref**(int *sock*, struct confd_identityref **rval*, const char **fmt*, ...);

int **cdb_get_ipv4_and_plen**(int *sock*, struct confd_ipv4_prefix **rval*, const char **fmt*, ...);

int **cdb_get_ipv6_and_plen**(int *sock*, struct confd_ipv6_prefix **rval*, const char **fmt*, ...);

int **cdb_get_dquad**(int *sock*, struct confd_dotted_quad **rval*, const char **fmt*, ...);

int **cdb_vget**(int *sock*, confd_value_t **v*, const char **fmt*, va_list *args*);

int **cdb_get_object**(int *sock*, confd_value_t **values*, int *n*, const char **fmt*, ...);

int **cdb_get_objects**(int *sock*, confd_value_t **values*, int *n*, int *ix*, int *nobj*, const char **fmt*, ...);

int **cdb_get_values**(int *sock*, confd_tag_value_t **values*, int *n*, const char **fmt*, ...);

```
int cdb_set_elem(int sock, confd_value_t *val, const char *fmt, ...);

int cdb_set_elem2(int sock, const char *strval, const char *fmt, ...);

int cdb_vset_elem(int sock, confd_value_t *val, const char *fmt, va_list
args);

int cdb_set_case(int sock, const char *choice, const char *scase, const
char *fmt, ...);

int cdb_create(int sock, const char *fmt, ...);

int cdb_delete(int sock, const char *fmt, ...);

int cdb_set_object(int sock, const confd_value_t *values, int n, const
char *fmt, ...);

int cdb_set_values(int sock, const confd_tag_value_t *values, int n,
const char *fmt, ...);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to the ConfD built-in XML database, CDB. The purpose of this API is to provide a read and subscription API to CDB.

CDB owns and stores the configuration data and the user of the API wants to read that configuration data and also get notified when someone through either NETCONF, SNMP, the CLI, the Web UI or the MAAPI modifies the data so that the application can re-read the configuration data and act accordingly.

CDB can also store operational data, i.e. data which is designated with a `"config false"` statement in the YANG data model. Operational data can be both read and written by the applications, but NETCONF and the other northbound agents can only read the operational data.

PATHS

The majority of the functions described here take as their two last arguments a format string and a variable number of extra arguments as in: char *`fmt`, `...`);

The `fmt` is a printf style format string which is used to format a path into the XML data tree. Assume the following YANG fragment:

```
container hosts {
  list host {
    key name;
    leaf name {
      type string;
    }
    leaf domain {
      type string;
    }
```

```
    leaf defgw {
      type inet:ipv4-address;
    }
    container interfaces {
      list interface {
        key name;
        leaf name {
          type string;
        }
        leaf ip {
          type inet:ipv4-address;
        }
        leaf mask {
          type inet:ipv4-address;
        }
        leaf enabled {
          type boolean;
        }
      }
    }
  }
}
```

Furthermore, assuming our database is populated with the following data.

```
<hosts xmlns="http://example.com/ns/hst/1.0">
  <host>
    <name>buzz</name>
    <domain>tail-f.com</domain>
    <defgw>192.168.1.1</defgw>
    <interfaces>
      <interface>
        <name>eth0</name>
        <ip>192.168.1.61</ip>
        <mask>255.255.255.0</mask>
        <enabled>true</enabled>
      </interface>
      <interface>
        <name>eth1</name>
        <ip>10.77.1.44</ip>
        <mask>255.255.0.0</mask>
        <enabled>false</enabled>
      </interface>
    </interfaces>
  </host>
</hosts>
```

The format path `/hosts/host{buzz}/defgw` refers to the leaf called `defgw` of the host whose key (`name` leaf) is `buzz`.

The format path `/hosts/host{buzz}/interfaces/interface{eth0}/ip` refers to the leaf called `ip` in the `eth0` interface of the host called `buzz`.

It is possible loop through all entries in a list as in:

```
n = cdb_num_instances(sock, "/hosts/host");
for (i=0; i<n; i++) {
    cdb_cd(sock, "/hosts/host[%d]", i)
    .....
```

Thus instead of an actually instantiated key inside a pair of curly braces `{key}`, we can use a temporary integer key inside a pair of brackets `[n]`.

We can use the following modifiers:

%d requiring an integer parameter (type int) to be substituted.

%u requiring an unsigned integer parameter (type unsigned int) to be substituted.

%s requiring a char* string parameter to be substituted.

%ip4 requiring a struct in_addr* to be substituted.

%ip6 requiring a struct in6_addr* to be substituted.

%x requiring a confd_value_t* to be substituted.

%*x requiring an array length and a confd_value_t* pointing to an array of values to be substituted.

%h requiring a confd_hkeypath_t* to be substituted.

%*h requiring a length and a confd_hkeypath_t* to be substituted.

Thus,

```
char *hname = "earth";
struct in_addr ip;
ip.s_addr = inet_addr("127.0.0.1");

cdb_cd(sock, "/hosts/host{%s}/bar{%ip4}", hname, &ip);
```

would change the current position to the path: `"/hosts/host{earth}/bar{127.0.0.1}"`

It is also possible to use the different '%' modifiers outside the curly braces, thus the above example could have been written as:

```
char *prefix = "/hosts/host";
cdb_cd(sock, "%s{%s}/bar{%ip4}", prefix, hname, &ip);
```

If an element has multiple keys, the keys must be space separated as in `cdb_cd("/bars/bar{%s %d}/item", str, i);`. However the '%*x' modifier is an exception to this rule, and it is especially useful when we have a number of key values that are unknown at compile time. If we have a list `foo` which is known to have two keys, and we have those keys in an array `key[]`, we can use `cdb_cd("/foo{%x %x}", &key[0], &key[1]);`. But if the number of keys is unknown at compile time (or if we just want a more compact code), we can instead use `cdb_cd("/foo{%*x}", n, key);` where *n* is the number of keys.

The '%h' and '%*h' modifiers can only be used at the beginning of a format path, as they expand to the absolute path corresponding to the confd_hkeypath_t. These modifiers are particularly useful with `cdb_diff_iterate()` (see below), or for MAAPI access in data provider callbacks (see confd_lib_maapi(3) and confd_lib_dp(3)). The '%*h' variant allows for using only the initial part of a confd_hkeypath_t, as specified by the preceding length argument (similar to '%.*s' for `printf(3)`).

For example, if the `iter()` function passed to `cdb_diff_iterate()` has been invoked with a confd_hkeypath_t *kp that corresponds to `/hosts/host{buzz}`, we can read the `defgw` child element with

```
confd_value_t v;
cdb_get(s, &v, "%h/defgw", kp);
```

or the entire list entry with

```
confd_value_t v[5];
cdb_get_object(sock, v, 5, "%h", kp);
```

or the `defgw` child element for host `mars` with

```
confd_value_t v;
cdb_get(s, &v, "%*h{mars}/defgw", kp->len - 1, kp);
```

All the functions that take a path on this form also have a va_list variant, of the same form as `cdb_vget()` and `cdb_vset_elem()`, which are the only ones explicitly documented below. I.e. they have a prefix "cdb_v" instead of "cdb_", and take a single va_list argument instead of a variable number of arguments.

FUNCTIONS

All functions return CONFD_OK (0), CONFD_ERR (-1) or CONFD_EOF (-2) unless otherwise stated. CONFD_EOF means that the socket to ConfD has been closed.

Whenever CONFD_ERR is returned from any API function described here, it is possible to obtain additional information on the error through the symbol `confd_errno`, see the ERRORS section in the confd_lib_lib(3) manual page.

`int` **`cdb_connect`**`(int` *`sock`*`, enum cdb_sock_type` *`type`*`, const struct sockaddr*` *`srv`*`, int` *`srv_sz`*`);`

The application has to connect to ConfD before it can interact. There are two different types of connections identified by cdb_sock_type:

`CDB_DATA_SOCKET`	This is a socket which is used to read configuration data, or to read and write operational data.
`CDB_SUBSCRIPTION_SOCKET`	This is a socket which is used to receive notifications about updates to the database. A subscription socket needs to be part of the application poll set.

Additionally the type CDB_READ_SOCKET is accepted for backwards compatibility - it is equivalent to CDB_DATA_SOCKET.

A call to `cdb_connect()` is typically followed by a call to either `cdb_start_session()` for a reading session or a call to `cdb_subscribe()` for a subscription socket.

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_connect_name`**`(int` *`sock`*`, enum cdb_sock_type` *`type`*`, const struct sockaddr*` *`srv`*`, int` *`srv_sz`*`, const char *`*`name`*`);`

When we use `cdb_connect()` to create a connection to ConfD/CDB, the *`name`* parameter passed to the library initialization function `confd_init()` (see confd_lib_lib(3)) is used to identify the connection in status reports and logs. I we want different names to be used for different connections from the same application process, we can use `cdb_connect_name()` with the wanted name instead of `cdb_connect()`.

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

```
int cdb_mandatory_subscriber(int sock, const char *name);
```

Attaches a mandatory attribute and a mandatory name to the subscriber identified by *sock*. The *name* parameter is distinct from the name parameter in `cdb_connect_name`.

CDB keeps a list of mandatory subscribers for infinite extent, i.e. until confd is restarted. The function is idempotent.

Absence of one or more mandatory subscribers will result in abort of all transactions. A mandatory subscriber must be present during the entire PREPARE delivery phase.

If a mandatory subscriber crash during a PREPARE delivery phase, the subscriber should be restarted and the commit operation should be retried.

A mandatory subscriber is present if the subscriber has issed at least one `cdb_subscribe()` call followed by a `cdb_subscribe_done()` call.

A call to `cdb_mandatory_subscriber()` is only allowed before `cdb_subscribe_done()` has been called.

Note

Only applicable for two-phase subscribers.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

```
int cdb_set_namespace(int sock, int hashed_ns);
```

If we want to access data in CDB where the toplevel element name is not unique, we need to set the namespace. We are reading data related to a specific .fxs file. confdc can be used to generate a .h file with a #define for the namespace, by the flag `--emit-h` to confdc (see confdc(1)).

It is also possible to indicate which namespace to use through the namespace prefix when we read and write data. Thus the path `/foo:bar/baz` will get us `/bar/baz` in the namespace with prefix "foo" regardless of what the "set" namespace is. And if there is only one toplevel element called "bar" across all namespaces, we can use `/bar/baz` without the prefix and without calling `cdb_set_namespace()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int cdb_end_session(int sock);
```

We use `cdb_connect()` to establish a read socket to CDB. When the socket is closed, the read session is ended. We can reuse the same socket for another read session, but we must then end the session and create another session using `cdb_start_session()`.

While we have a live CDB read session for configuration data, CDB is normally locked for writing. Thus all external entities trying to modify CDB are blocked as long as we have an open CDB read session. It is very important that we remember to either `cdb_end_session()` or `cdb_close()` once we have read what we wish to read.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`cdb_start_session`**`(int` *`sock`*`, enum cdb_db_type` *`db`*`);`

Starts a new session on an already established socket to CDB. The db parameter should be one of:

`CDB_RUNNING` Creates a read session towards the running database.

`CDB_PRE_COMMIT_RUNNING` Creates a read session towards the running database as it was before the current transaction was committed. This is only possible between a subscription notification and the final `cdb_sync_subscription_socket()`. At any other time trying to call `cdb_start_session()` will fail with confd_errno set to CONFD_ERR_NOEXISTS.

In the case of a `CDB_SUB_PREPARE` subscription notification a session towards `CDB_PRE_COMMIT_RUNNING` will (in spite of the name) will return values as they were *before the transaction which is about to be committed* took place. This means that if you want to read the new values during a `CDB_SUB_PREPARE` subscription notification you need to create a session towards `CDB_RUNNING`. However, since it is locked the session needs to be started in lockless mode using `cdb_start_session2()`. So for example:

```
cdb_read_subscription_socket2(ss, &type, &flags, &subp, &len);
/* ... */
switch (type) {
case CDB_SUB_PREPARE:
    /* Set up a lockless session to read new values: */
    cdb_start_session2(s, CDB_RUNNING, 0);
    read_new_config(s);
    cdb_end_session(s);
    cdb_sync_subscription_socket(ss, CDB_DONE_PRIORITY);
    break;
    /* ... */
```

`CDB_STARTUP` Creates a read session towards the startup database.

`CDB_OPERATIONAL` Creates a read/write session towards the operational database. For further details about working with operational data in CDB, see the `OPERATIONAL DATA` section below.

Note

Subscriptions on operational data will not be triggered from a session created with this function - to trigger operational data subscriptions, we need to use `cdb_start_session2()`, see below.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_LOCKED, CONFD_ERR_NOEXISTS

If the error is CONFD_ERR_LOCKED it means that we are trying to create a new CDB read session precisely when the write phase of some transaction is occurring. Thus correct usage of `cdb_start_session()` is:

```
while (1) {
```

```
   if (cdb_start_session(sock, CDB_RUNNING) == CONFD_OK)
      break;
   if (confd_errno == CONFD_ERR_LOCKED) {
      sleep(1);
      continue;
   }
   .... handle error
}
```

Alternatively we can use `cdb_start_session2()` with *`flags`* = CDB_LOCK_SESSION|CDB_LOCK_WAIT. This means that the call will block until the lock has been acquired, and thus we do not need the retry loop.

`int` **`cdb_start_session2`**`(int` *`sock`*`, enum cdb_db_type` *`db`*`, int` *`flags`*`);`

This function may be used instead of `cdb_start_session()` if it is considered necessary to have more detailed control over some aspects of the CDB session - if in doubt, use `cdb_start_session()` instead. The *`sock`* and *`db`* arguments are the same as for `cdb_start_session()`, and these values can be used for *`flags`* (ORed together if more than one):

```
#define CDB_LOCK_WAIT     (1 << 0)
#define CDB_LOCK_SESSION  (1 << 1)
#define CDB_LOCK_REQUEST  (1 << 2)
#define CDB_LOCK_PARTIAL  (1 << 3)
```

The flags affect sessions for the different database types as follows:

`CDB_RUNNING` — CDB_LOCK_SESSION obtains a read lock for the complete session, i.e. using this flag alone is equivalent to calling `cdb_start_session()`. CDB_LOCK_REQUEST obtains a read lock only for the duration of each read request. This means that values of elements read in different requests may be inconsistent with each other, and the consequences of this must be carefully considered. In particular, the use of `cdb_num_instances()` and the `[n]` "integer index" notation in keypaths is inherently unsafe in this mode. Note: The implementation will not actually obtain a lock for a single-value request, since that is an atomic operation anyway. The CDB_LOCK_PARTIAL flag is not allowed.

`CDB_STARTUP` — Same as CDB_RUNNING.

`CDB_PRE_COMMIT_RUNNING` — This database type does not have any locks, which means that it is an error to call `cdb_start_session2()` with any CDB_LOCK_XXX flag included in *`flags`*. Using a *`flags`* value of 0 is equivalent to calling `cdb_start_session()`.

`CDB_OPERATIONAL` — CDB_LOCK_REQUEST obtains a "subscription lock" for the duration of each write request. This can be described as an "advisory exclusive" lock, i.e. only one client at a time can hold the lock (unless CDB_LOCK_PARTIAL is used), but the lock does not affect clients that do not attempt to obtain it. It also does not affect the reading of operational data. The purpose of this lock is to indicate that the client wants the write operation to generate subscription notifications. The lock remains in effect until any/all subscription notifications generated as a result of the write has been delivered.

If the CDB_LOCK_PARTIAL flag is used together with CDB_LOCK_REQUEST, the "subscription lock" only applies to the smallest data subtree that includes all the data in the write request. This means that multiple writes that generates subscription notifications, and delivery of the corresponding notifications, can proceed in parallel as long as they affect disjunct parts of the data tree.

The CDB_LOCK_SESSION flag is not allowed. Using a *`flags`* value of 0 is equivalent to calling `cdb_start_session()`.

In all cases of using CDB_LOCK_SESSION or CDB_LOCK_REQUEST described above, adding the CDB_LOCK_WAIT flag means that instead of failing with CONFD_ERR_LOCKED if the lock can not be obtained immediately, requests will wait for the lock to become available. When used with CDB_LOCK_SESSION it pertains to `cdb_start_session2()` itself, with CDB_LOCK_REQUEST it pertains to the individual requests.

While it is possible to use this function to start a session towards a configuration database type with no locking at all (*`flags`* = 0), this is strongly discouraged in general, since it means that even the values read in a single multi-value request (e.g. `cdb_get_object()`, see below) may be inconsistent with each other. However it is necessary to do this if we want to have a session open during semantic validation, see the "Semantic Validation" chapter in the User Guide - and in this particular case it is safe, since the transaction lock prevents changes to CDB during validation.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_LOCKED, CONFD_ERR_NOEXISTS, CONFD_ERR_PROTOUSAGE

`int` **`cdb_close`**`(int` *`sock`*`);`

Closes the socket. `cdb_end_session()` should be called before calling this function.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

Even if the call returns an error, the socket will be closed.

`int` **`cdb_wait_start`**`(int` *`sock`*`);`

This call waits until CDB has completed start-phase 1 and is available, when it is CONFD_OK is returned. If CDB already is available (i.e. start-phase >= 1) the call returns immediately. This can be used by a CDB client who is not synchronously started and only wants to wait until it can read its configuration. The call can be used after cdb_connect().

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_get_phase`**`(int` *`sock`*`, struct cdb_phase *`*`phase`*`);`

Returns the start-phase CDB is currently in, in the struct cdb_phase pointed to by the second argument. Also if CDB is in phase 0 and has initiated an init transaction (to load any init files) the flag CDB_FLAG_INIT is set in the flags field of struct cdb_phase and correspondingly if an upgrade session is started the CDB_FLAG_UPGRADE is set. The call can be used after cdb_connect() and returns CONFD_OK.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_initiate_journal_compaction`**`(int` *`sock`*`);`

Normally CDB handles journal compaction of the config datastore automatically. If this has been turned off (in the configuration file) then the A.cdb file will grow indefinitely unless this API function is called

periodically to initiate compaction. This function initiates a compaction and returns immediately (if the datastore is locked, the compaction will be delayed, but eventually compaction will take place). Calling this function when journal compaction is configured to be automatic has no effect.

Errors: -

```
int cdb_get_txid(int sock, struct cdb_txid *txid);
```

Read the last transaction id from CDB. This function can be used if we are forced to reconnect to CDB, If the transaction id we read is identical to the last id we had prior to loosing the CDB sockets we don't have to reload our managed object data. See the User Guide for full explanation. Returns CONFD_OK on success and CONFD_ERR or CONFD_EOF on failure.

```
int cdb_get_replay_txids(int sock, struct cdb_txid **txid, int *re-
sultlen);
```

When the subscriptionReplay functionality is enabled in confd.conf this function returns the list of available transactions that CDB can replay. The current transaction id will be the first in the list, the second at txid[1] and so on. The number of transactions is returned in `resultlen`. In case there are no replay transactions available (the feature isn't enabled or there hasn't been any transactions yet) only one (the current) transaction id is returned. It is up to the caller to `free()` `txid` when it is no longer needed.

```
int cdb_set_timeout(int sock, int timeout_secs);
```

A timeout for client actions can be specified via `/confdConfig/cdb/clientTimeout` in `confd.conf`, see the confd.conf(5) manual page. This function can be used to dynamically extend (or shorten) the timeout for the current action. Thus it is possible to configure a restrictive timeout in `confd.conf`, but still allow specific actions to have a longer execution time.

The function can be called either with a subscription socket during subscription delivery on that socket (including from the `iter()` function passed to `cdb_diff_iterate()`), or with a data socket that has an active session. The timeout is given in seconds from the point in time when the function is called.

Note

The timeout for subscription delivery is common for all the subscribers receiving notifications at a given priority. Thus calling the function during subscription delivery changes the timeout for all the subscribers that are currently processing notifications.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE, CONFD_ERR_BADSTATE

```
int cdb_exists(int sock, const char *fmt, ...);
```

Leafs in the data model may be optional, and presence containers and list entries may or may not exist. This function checks whether a node exists in CDB. Returns 0 for false, 1 for true and CONFD_ERR or CONFD_EOF for errors.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

```
int cdb_cd(int sock, const char *fmt, ...);
```

Changes the working directory according to the format path. Note that this function can not be used as an existence test.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

`int` **`cdb_pushd`**`(int sock, const char *fmt, ...);`

Similar to `cdb_cd()` but pushes the previous current directory on a stack.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSTACK, CONFD_ERR_BADPATH

`int` **`cdb_popd`**`(int sock);`

Pops the top element from the directory stack and changes directory to previous directory.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSTACK

`int` **`cdb_getcwd`**`(int sock, size_t strsz, char *curdir);`

Returns the current position as previously set by `cdb_cd()`, `cdb_pushd()`, or `cdb_popd()` as a string path. Note that what is returned is a pretty-printed version of the internal representation of the current position, it will be the shortest unique way to print the path but it might not exactly match the string given to `cdb_cd()`. The buffer in *curdir will be NULL terminated, and no more characters than strsz-1 will be written to it.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_getcwd_kpath`**`(int sock, confd_hkeypath_t **kp);`

Returns the current position like `cdb_getcwd()`, but as a pointer to a hashed keypath instead of as a string. The hkeypath is dynamically allocated, and may further contain dynamically allocated elements. The caller must free the allocated memory, easiest done by calling `confd_free_hkeypath()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_num_instances`**`(int sock, const char *fmt, ...);`

Returns the number of entries in a list. On error CONFD_ERR or CONFD_EOF is returned.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

`int` **`cdb_next_index`**`(int sock, const char *fmt, ...);`

Given a path to a list entry `cdb_next_index()` returns the position (starting from 0) of the next entry (regardless of whether the path exists or not). When the list has multiple keys a `*` may be used for the last keys to make the path partially instantiated. For example if `/foo/bar` has three integer keys, the following pseudo code could be used to iterate over all entries with `42` as the first key:

```
/* find the first entry of /foo/bar with 42 as first key */
ix = cdb_next_index(sock, "/foo/bar{42 * *}");
for (; ix>=0; ix++) {
    int32_t k1 = 0;
    cdb_get_int32(sock, &k1, "/foo/bar[%d]/key1", ix);
    if (k1 != 42) break;
    /* ... do something with /foo/bar[%d] ... */
}
```

If there is no next entry -1 is returned. It is not possible to use this function on an ordered-by user list. On error CONFD_ERR or CONFD_EOF is returned.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

`int` **`cdb_index`**`(int sock, const char *fmt, ...);`

Given a path to a list entry `cdb_index()` returns its position (starting from 0). On error CONFD_ERR or CONFD_EOF is returned.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

`int` **`cdb_is_default`**`(int` *`sock`*`, const char *`*`fmt`*`, ...);`

This function returns 1 for a leaf which has a default value defined in the data model when no value has been set, i.e. when the default value is in effect. It returns 0 for other existing leafs, and CONFD_ERR or CONFD_EOF for errors. There is normally no need to call this function, since CDB automatically provides the default value as needed when cdb_get() etc is called.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

`int` **`cdb_subscribe`**`(int` *`sock`*`, int` *`priority`*`, int` *`nspace`*`, int *`*`spoint`*`, const char *`*`fmt`*`, ...);`

Sets up a CDB subscription so that we are notified when CDB configuration data changes. There can be multiple subscription points from different sources, that is a single client daemon can have many subscriptions and there can be many client daemons.

Each subscription point is defined through a path similar to the paths we use for read operations. We can subscribe either to specific leafs or entire subtrees. Subscribing to list entries can be done using fully qualified paths, or tagpaths to match multiple entries. A path which isn't a leaf element automatically matches the subtree below that path. When specifying keys to a list entry it is possible to use the wildcard character * which will match any key value.

When subscribing to a leaf with a `tailf:default-ref` statement, or to a subtree with elements that have `tailf:default-ref`, implicit subscriptions to the referred leafs are added. This means that a change in a referred leaf will generate a notification for the subscription that has referring leaf(s) - but currently such a change will not be reported by `cdb_diff_iterate()`. Thus to get the new "effective" value of a referring leaf in this case, it is necessary to either read the value of the leaf with e.g. `cdb_get()` - or to use a subscription that includes the referred leafs, and use `cdb_diff_iterate()` when a notification for that subscription is received.

Some examples

`/hosts`	Means that we subscribe to any changes in the subtree - rooted at `/hosts`. This includes additions or removals of `host` entries as well as changes to already existing `host` entries.
`/hosts/host{www}/interfaces/interface{eth0}/ip`	Means we are notified when host `www` changes its IP address on `eth0`.
`/hosts/host/interfaces/interface/ip`	Means we are notified when any host changes any of its IP addresses.
`/hosts/host/interfaces`	Means we are notified when either an interface is added/removed or when an individual leaf element in an existing interface is changed.

The *`priority`* value is an integer. When CDB is changed, the change is performed inside a transaction. Either a **commit** operation from the CLI or a **candidate-commit** operation in NETCONF means that the running database is changed. These changes occur inside a ConfD transaction. CDB will handle the subscriptions in lock-step priority order. First all subscribers at the lowest priority are handled, once they all have replied and synchronized through calls to `cdb_sync_subscription_socket()` the next

set - at the next priority level is handled by CDB. Priority numbers are global, i.e. if there are multiple client daemons notifications will still be delivered in priority order per all subscriptions, not per daemon.

See `cdb_diff_iterate()` and cdb_diff_match() for ways of filtering subscription notifications and finding out what changed. The easiest way is though to not use either of the two above mentioned diff function but to solely rely on the positioning of the subscription points in the tree to figure out what changed.

`cdb_subscribe()` returns a *`subscription point`* in the return parameter *`spoint`*. This integer value is used to identify this particular subscription.

Because there can be many subscriptions on the same socket the client must notify ConfD when it is done subscribing and ready to receive notifications. This is done using `cdb_subscribe_done()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

`int` **`cdb_oper_subscribe`**`(int` *`sock`*`, int` *`nspace`*`, int *`*`spoint`*`, const char *`*`fmt`*`, ...);`

Sets up a CDB subscription for changes in the operational data base. Similar to the subscriptions for configuration data, we can be notified of changes to the operational data stored in CDB. Note that there are several differences from the subscriptions for configuration data:

- Notifications are only generated if the writer has taken a subscription lock, see `cdb_start_session2()` above.
- Priorities are not used for these notifications.
- It is not possible to receive the previous value for modified leafs in `cdb_diff_iterate()`.
- A special synchronization reply must be used when the notifications have been read (see `cdb_sync_subscription_socket()` below).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

`int` **`cdb_subscribe2`**`(int` *`sock`*`, enum cdb_sub_type` *`type`*`, int` *`flags`*`, int` *`priority`*`, int *`*`spoint`*`, int` *`nspace`*`, const char *`*`fmt`*`, ...);`

This function supersedes the current `cdb_subscribe()` and `cdb_oper_subscribe()` as well as makes it possible to use the new two phase subscription method. The cdb_sub_type is defined as:

```
enum cdb_sub_type {
    CDB_SUB_RUNNING = 1,
    CDB_SUB_RUNNING_TWOPHASE = 2,
    CDB_SUB_OPERATIONAL = 3
};
```

The CDB subscription type `CDB_SUB_RUNNING` is the same as `cdb_subscribe()`, `CDB_SUB_OPERATIONAL` is the same as `cdb_oper_subscribe()`, and `CDB_SUB_RUNNING_TWOPHASE` does a two phase subscription.

The flags argument should be set to 0, or a combination of:

`CDB_SUB_WANT_ABORT_ON_ABORT` Normally if a subscriber is the one to abort a transaction it will not receive an abort notification. This flags means that this subscriber wants an abort notification even if it was the one that called

cdb_sub_abort_trans(). This flag is only valid when the subscription type is CDB_SUB_RUNNING_TWOPHASE.

The two phase subscriptions work like this: A subscriber uses cdb_subscribe2() with the type set to CDB_SUB_RUNNING_TWOPHASE to register as many subscription points as required. The cdb_subscribe_done() function is used to indicate that no more subscription points will be registered on that particular socket. Only after cdb_subscribe_done() is called will subscription notifications be delivered.

Once a transaction enters prepare state all CDB two phase subscribers will be notified in priority order (lowest priority first, subscribers with the same priority is delivered in parallel). The cdb_read_subscription_socket2() function will set type to CDB_SUB_PREPARE. Once all subscribers have acknowledged the notification by using the function cdb_sync_subscription_socket(CDB_DONE_PRIORITY) they will subsequently be notified when the transaction is committed. The CDB_SUB_COMMIT notification is the same as the current subscription mechanism, so when a transaction is committed all subscribers will be notified (again in priority order).

When a transaction is aborted, delivery of any remaining CDB_SUB_PREPARE notifications is cancelled. The subscribers that had already been notified with CDB_SUB_PREPARE will be notified with CDB_SUB_ABORT (This notification will be done in reverse order of the CDB_SUB_PREPARE notification). The transaction could be aborted because one of the subscribers that received CDB_SUB_PREPARE called cdb_sub_abort_trans(), but it could also be caused for other reasons, for example another data provider (than CDB) can abort the transaction.

Note

Two phase subscriptions are not supported for NCS.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

```
int cdb_subscribe_done(int sock);
```

When a client is done registering all its subscriptions on a particular subscription socket it must call cdb_subscribe_done(). No notifications will be delivered until then.

```
int cdb_trigger_subscriptions(int sock, int sub_points[], int len);
```

This function makes it possible to trigger CDB subscriptions for configuration data even though the configuration has not been modified. The caller will trigger all subscription points passed in the sub_points array (or all subscribers if the array is of zero length) in priority order, and the call will not return until the last subscriber has called cdb_sync_subscription_socket().

The call is blocking and doesn't return until all subscribers have acknowledged the notification. That means that it is not possible to use cdb_trigger_subscriptions() in a cdb subscriber process (without forking a process or spawning a thread) since it would cause a deadlock.

The subscription notification generated by this "synthetic" trigger will seem like a regular subscription notification to a subscription client. As such, it is possible to use cdb_diff_iterate() to traverse the changeset. CDB will make up this changeset in which all leafs in the configuration will appear to be set, and all list entries and presence containers will appear as if they are created.

If the client is a two-phase subscriber, a prepare notification will first be delivered and if any client aborts this synthetic transaction further delivery of subscription notification is suspended and an error is returned to the caller of cdb_trigger_subscriptions(). The error is the result of mapping the

CONFD_ERRCODE as set by the aborting client as described for MAAPI in the EXTENDED ERROR REPORTING section in the confd_lib_lib(3) manpage. Note however that the configuration is still the way it is - so it is up to the caller of `cdb_trigger_subscriptions()` to take appropriate action (for example: raising an alarm, restarting a subsystem, or even rebooting the system).

If one or more subscription ids is passed in the subids array that are not valid, an error (`CONFD_ERR_PROTOUSAGE`) will be returned and no subscriptions will be triggered. If no subscription ids are passed this error can not occur (even if there aren't any subscribers).

`int` **`cdb_trigger_oper_subscriptions`**`(int` *`sock`*`, int` *`sub_points[]`*`, int` *`len`*`, int` *`flags`*`);`

This function works like `cdb_trigger_subscriptions()`, but for CDB subscriptions to operational data. The caller will trigger all subscription points passed in the *`sub_points`* array (or all operational data subscribers if the array is of zero length), and the call will not return until the last subscriber has called cdb_sync_subscription_socket().

Since the generation of subscription notifications for operational data requires that the subscription lock is taken (see `cdb_start_session2()`), this function implicitly attempts to take a "global" subscription lock. If the subscription lock is already taken, the function will by default return CONFD_ERR with `confd_errno` set to CONFD_ERR_LOCKED. To instead have it wait until the lock becomes available, CDB_LOCK_WAIT can be passed for the *`flags`* parameter.

`int` **`cdb_replay_subscriptions`**`(int` *`sock`*`, struct cdb_txid *`*`txid`*`, int` *`sub_points[]`*`, int` *`len`*`);`

This function makes it possible to replay the subscription events for the last configuration change to some or all CDB subscribers. This call is useful in a number of recovery scenarios, where some CDB subscribers lost connection to ConfD before having received all the changes in a transaction. The replay functionality is only available if it has been enabled in confd.conf

The caller specifies the transaction id of the last transaction that the application has completely seen and acted on. This verifies that the application has only missed (part of) the last transaction. If a different (older) transaction ID is specified, an error is returned and no subscriptions will be triggered. If the transaction id is the latest transaction ID (i.e. the caller is already up to date) nothing is triggered and CONFD_OK is returned.

By calling this function, the caller will potentially trigger all subscription points passed in the sub_points array (or all subscribers if the array is of zero length). The subscriptions will be triggered in priority order, and the call will not return until the last subscriber has called cdb_sync_subscription_socket().

The call is blocking and doesn't return until all subscribers have acknowledged the notification. That means that it is not possible to use `cdb_replay_subscriptions()` in a cdb subscriber process (without forking a process or spawning a thread) since it would cause a deadlock.

The subscription notification generated by this "synthetic" trigger will seem like a regular subscription notification to a subscription client. It is possible to use `cdb_diff_iterate()` to traverse the changeset.

If the client is a two-phase subscriber, a prepare notification will first be delivered and if any client aborts this synthetic transaction further delivery of subscription notification is suspended and an error is returned to the caller of `cdb_replay_subscriptions()`. The error is the result of mapping the CONFD_ERRCODE as set by the aborting client as described for MAAPI in the EXTENDED ERROR REPORTING section in the confd_lib_lib(3) manpage.

`int` **`cdb_read_subscription_socket`**`(int` *`sock`*`, int` *`sub_points[]`*`, int *`*`resultlen`*`);`

The subscription socket - which is acquired through a call to `cdb_connect()` - must be part of the application poll set. Once the subscription socket has I/O ready to read, we must call `cdb_read_subscription_socket()` on the subscription socket.

The call will fill in the result in the array *`sub_points`* with a list of integer values containing *subscription points* earlier acquired through calls to `cdb_subscribe()`. The global variable `cdb_active_subscriptions` can be read to find how many active subscriptions the application has. Make sure the `sub_points[]` array is at least this big, otherwise the confd library will write in unallocated memory.

The subscription points may be either for configuration data or operational data (if `cdb_oper_subscribe()` has been used on the same socket), but they will all be of the same "type" - i.e. a single call of the function will never deliver a mix of configuration and operational data subscription points.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_read_subscription_socket2`**`(int` *`sock`*`, enum cdb_sub_notification *`*`type`*`, int *`*`flags`*`, int *`*`subpoints[]`*`, int *`*`resultlen`*`);`

```
enum cdb_sub_notification {
    CDB_SUB_PREPARE = 1,
    CDB_SUB_COMMIT = 2,
    CDB_SUB_ABORT = 3,
    CDB_SUB_OPER = 4
};
```

This is another version of the `cdb_read_subscription_socket()` with two important differences:

1. In this version *subpoints is allocated by the library*, and it is up to the caller of this function to `free()` it when it is done.

2. It is possible to retrieve the type of the subscription notification via the *`type`* return parameter.

All parameters except *`sock`* are return parameters. It is legal to pass in *`flags`* and *`type`* as `NULL` pointers (in which case type and flags cannot be retrieved). *`subpoints`* is an array of integers, the length is indicated in *`resultlen`*, it is allocated by the library, and *must be freed by the caller*. The *`type`* parameter is what the subscriber uses to distinguish the different types of subscription notifications.

The *`flags`* return parameter can have the following bits set:

`CDB_SUB_FLAG_IS_LAST`	This bit is set when this notification is the last of its type for this subscription socket.
`CDB_SUB_FLAG_HA_IS_SLAVE`	This bit is set when ConfD runs in HA mode, and the current HA mode is slave. I.e. it is a convenient way for the subscriber to know wether this node is in slave mode or not.
`CDB_SUB_FLAG_TRIGGER`	This bit is set when the cause of the subscription notification is that someone called `cdb_trigger_subscriptions()`.
`CDB_SUB_FLAG_REVERT`	If a confirming commit is aborted it will look to the CDB subscriber as if a transaction happened that is the reverse of what the original transaction was. This bit will be set when such a transaction is the cause of the notification. Note that for a two-phase subscriber both a prepare and a commit notification is delivered. However it is not possible to reply by calling `cdb_sub_abort_trans()` for the

prepare notification in this case, instead the subscriber will have to take appropriate backup action if it needs to abort (for example: raise an alarm, restart, or even reboot the system).

CDB_SUB_FLAG_HA_SYNC — This bit is set when the cause of the subscription notification is initial synchronization of a HA slave from CDB on the master.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

```
int cdb_diff_iterate(int sock, int subid, enum cdb_iter_ret (*iter)
(confd_hkeypath_t *kp, enum cdb_iter_op op, confd_value_t *oldv,
confd_value_t *newv, void *state), int flags, void *initstate);
```

After reading the subscription socket the `cdb_diff_iterate()` function can be used to iterate over the changes made in CDB data that matched the particular subscription point given by *`subid`*.

The user defined function `iter()` will be called for each element that has been modified and matches the subscription. The `iter()` callback receives the *`confd_hkeypath_t kp`* which uniquely identifies which node in the data tree that is affected, the operation, and optionally the values it has before and after the transaction. The *`op`* parameter gives the modification as:

MOP_CREATED — The list entry, `presence` container, or leaf of type `empty` given by *`kp`* has been created.

MOP_DELETED — The list entry, `presence` container, or optional leaf given by *`kp`* has been deleted.

If the subscription was triggered because an ancestor was deleted, the `iter()` function will not called at all if the delete was above the subscription point. However if the flag ITER_WANT_ANCESTOR_DELETE is passed to `cdb_diff_iterate()` then deletes that trigger a descendant subscription will also generate a call to `iter()`, and in this case *`kp`* will be the path that was actually deleted.

MOP_MODIFIED — A descendant of the list entry given by *`kp`* has been modified.

MOP_VALUE_SET — The value of the leaf given by *`kp`* has been set to *`newv`*.

MOP_MOVED_AFTER — The list entry given by *`kp`*, in an `ordered-by user` list, has been moved. If *`newv`* is NULL, the entry has been moved first in the list, otherwise it has been moved after the entry given by *`newv`*. In this case *`newv`* is a pointer to an array of key values identifying an entry in the list. The array is terminated with an element that has type C_NOEXISTS.

By setting the *`flags`* parameter ITER_WANT_REVERSE two-phase subscribers may use this function to traverse the reverse changeset in case of CDB_SUB_ABORT notification. In this scenario a two-phase subscriber traverses the changes in the prepare phase (CDB_SUB_PREPARE notification) and if the transaction is aborted the subscriber may iterate the inverse to the changes during the abort phase (CDB_SUB_PREPARE notification).

For configuration subscriptions, the previous value of the node can also be passed to `iter()` if the *`flags`* parameter contains ITER_WANT_PREV, in which case *`oldv`* will be pointing to it (otherwise NULL). For operational data subscriptions, the ITER_WANT_PREV flag is ignored, and *`oldv`* is always NULL - there is no equivalent to CDB_PRE_COMMIT_RUNNING that holds "old" operational data.

If `iter()` returns ITER_STOP, no more iteration is done, and CONFD_OK is returned. If `iter()` returns ITER_RECURSE iteration continues with all children to the node. If `iter()` returns

ITER_CONTINUE iteration ignores the children to the node (if any), and continues with the node's sibling, and if `iter()` returns ITER_UP the iteration is continued with the node's parents sibling. If, for some reason, the `iter()` function wants to return control to the caller of `cdb_diff_iterate()` *before* all the changes has been iterated over it can return ITER_SUSPEND. The caller then has to call `cdb_diff_iterate_resume()` to continue/finish the iteration.

The *`state`* parameter can be used for any user supplied state (i.e. whatever is supplied as *`initstate`* is passed as *`state`* to `iter()` in each invocation).

By default the traverse order is undefined but guaranteed to be the most efficient one. The traverse order may be changed by setting setting a bit in the *`flags`* parameter:

ITER_WANT_SCHEMA_ORDER The `iter()` function will be invoked in *schema* order (i.e. in the order in which the elements are defined in the YANG file).

ITER_WANT_LEAF_FIRST_ORDER The `iter()` function will be invoked for leafs first, then non-leafs.

ITER_WANT_LEAF_LAST_ORDER The `iter()` function will be invoked for non-leafs first, then leafs.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_BADSTATE, CONFD_ERR_PROTOUSAGE.

`int` **`cdb_diff_iterate_resume`**`(int` *`sock`*`, enum cdb_iter_ret` *`reply`*`, enum cdb_iter_ret (*`iter`*`)(confd_hkeypath_t *`*`kp`*`, enum cdb_iter_op` *`op`*`, confd_value_t *`*`oldv`*`, confd_value_t *`*`newv`*`, void *`*`state`*`), void *`*`resumestate`*`);`

The application *must* call this function whenever an iterator function has returned `ITER_SUSPEND` to finish up the iteration. If the application does not wish to continue iteration it must at least call `cdb_diff_iterate_resume(s, ITER_STOP, NULL, NULL);` to clean up the state. The *`reply`* parameter is what the iterator function would have returned (i.e. normally ITER_RECURSE or ITER_CONTINUE) if it hadn't returned ITER_SUSPEND. Note that it is up to the iterator function to somehow communicate that it has returned ITER_SUSPEND to the caller of `cdb_diff_iterate()`, this can for example be a field in a struct for which a pointer to can passed back and forth in the `state`/`resumestate` variable.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_BADSTATE.

`int` **`cdb_diff_match`**`(int` *`sock`*`, int` *`subid`*`, struct xml_tag` *`tags[]`*`, int` *`tagslen`*`);`

This function can be invoked when a subscription point has fired. Similar to the `confd_hkp_tagmatch()` function it takes an argument which is an array of XML tags. The function will invoke `cdb_diff_iterate()` on a subscription socket. Using combinations of `ITER_STOP`, `ITER_CONTINUE` and `ITER_RECURSE` return values, the function checks a tagpath and decides whether any changes (under the subscription point) has occurred that also match the provided path *`tags`*. It is slightly easier to use this function than `cdb_diff_iterate()` but can also be slower since it is a general purpose matcher.

If we have a subscription point at `/root`, we could invoke this function as:

```
struct xml_tag tags[] = {{root_root, root__ns},
                         {root_servers, root__ns},
                         {root_server, root__ns}};
/* /root/servers/server */
```

```
int retv = cdb_diff_match(subsock, subpoint, tags, 3);
```

The function returns 1 if there were any changes under *subpoint* that matched *tags*, 0 if no match was found and CONFD_ERR on error.

int **cdb_cli_diff_iterate**(int *sock*, int *subid*, cli_diff_iter_function_t **iter*, int *flags*, void **initstate*);

Where the cli_diff_iter_function_t is defined as:

```
typedef enum cdb_iter_ret
    (cli_diff_iter_function_t)(confd_hkeypath_t *kp,
                               enum cdb_iter_op op,
                               confd_value_t *oldv,
                               confd_value_t *newv,
                               char *clistr,
                               int token_count,
                               struct confd_cli_token *tokens,
                               void *state);
```

Note

This function is DEPRECATED. Use cdb_get_modifications_cli() instead.

The function cdb_cli_diff_iterate() works just like the cdb_diff_iterate() function, except the iter() function takes three additional parameters, *clistr*, *token_count*, and *tokens*. The *clistr* is a string containing the (C-style) rendering of the CLI commands equivalent to the current keypath/operation. The *tokens* is actually an array of length *token_count*, it contains the CLI string broken down by token.

The string and the token array (including all the strings in the array) are allocated by the library, and will be freed by the library when the *iter* function returns.

If *flags* has the ITER_WANT_SCHEMA_ORDER bit set, then the iter() function will be invoked in *schema* order (i.e. in the order in which the elements are defined in the YANG file). Normally the order is undefined, which is most efficient.

Note that the cli commands are independent of whether the originating request actually came in over the CLI or some other northbound interface.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_BADSTATE.

int **cdb_get_modifications**(int *sock*, int *subid*, int *flags*, confd_tag_value_t ***values*, int **nvalues*, const char **fmt*, ...);

The cdb_get_modifications() function can be called after reception of a subscription notification to retrieve all the changes that caused the subscription notification. The socket *s* is the subscription socket, the subscription id must also be provided. Optionally a path can be used to limit what is returned further (only changes below the supplied path will be returned), if this isn't needed fmt can be set to NULL.

When cdb_get_modifications() returns CONFD_OK, the results are in *values*, which is a tag value array with length *nvalues*. The library allocates memory for the results, which must be free:d by the caller. This can in all cases be done with code like this:

```
confd_tag_value_t *values;
```

```
int nvalues, i;

if (cdb_get_modifications(sock, subid, flags, &values, &nvalues,
                          "/some/path") == CONFD_OK) {
    ...
    for (i = 0; i < nvalues; i++)
        confd_free_value(CONFD_GET_TAG_VALUE(&values[i]));
    free(values);
}
```

The tag value array differs somewhat between how it is described in the confd_types(3) manual page, most notably only the values that were modified in this transaction are included. In addition to that these are the different values of the tags depending on what happened in the transaction:

- A leaf of type empty that has been deleted has the value of `C_NOEXISTS`, and when it is created it has the value `C_XMLTAG`.
- A leaf or a leaf-list that has been set to a new value (or its default value) is included with that new value. If the leaf or leaf-list is optional, then when it is deleted the value is `C_NOEXISTS`.
- Presence containers are included when they are created or when they have modifications below them (by the usual `C_XMLBEGIN`, `C_XMLEND` pair). If a presence container has been deleted its tag is included, but has the value `C_NOEXISTS`.

By default `cdb_get_modifications()` does not include list instances (created, deleted, or modified) - but if the `CDB_GET_MODS_INCLUDE_LISTS` flag is included in the *`flags`* parameter, list instances will be included. Created and modified instances are included wrapped in the `C_XMLBEGIN` / `C_XMLEND` pair, with the keys first. Deleted list instances instead begin with `C_XMLBEGINDEL`, then follows the keys, immediately followed by a `C_XMLEND`.

When processing a `CDB_SUB_ABORT` notification for a two phase subscription, it is also possible to request a list of "reverse" modifications instead of the normal "forward" list. This is done by including the `CDB_GET_MODS_REVERSE` flag in the *`flags`* parameter.

`int` **`cdb_get_modifications_iter`**`(int` *`sock`*`, int` *`flags`*`, confd_tag_value_t **`values`*`, int *`nvalues`*`);`

The `cdb_get_modifications_iter()` is basically a convenient short-hand of the `cdb_get_modifications()` function intended to be used from within a iteration function started by `cdb_diff_iterate()`. In this case no subscription id is needed, and the path is implicitly the current position in the iteration.

Combining this call with `cdb_diff_iterate()` makes it for example possible to iterate over a list, and for each list instance fetch the changes using `cdb_get_modifications_iter()`, and then return `ITER_CONTINUE` to process next instance.

Note

Note: The `CDB_GET_MODS_REVERSE` flag is ignored by `cdb_get_modifications_iter()`. It will instead return a "forward" or "reverse" list of modifications for a `CDB_SUB_ABORT` notification according to whether the `ITER_WANT_REVERSE` flag was included in the *`flags`* parameter of the `cdb_diff_iterate()` call.

`int` **`cdb_sync_subscription_socket`**`(int` *`sock`*`, enum cdb_subscription_sync_type` *`st`*`);`

Once we have read the subscription notification through a call to `cdb_read_subscription_socket()` and optionally used the `cdb_diff_iterate()` to iterate through the changes as well as acted on the changes to CDB, we must synchronize with CDB so that CDB can continue and deliver further subscription messages to subscribers with higher priority numbers.

There are four different types of synchronization replies the application can use in the enum cdb_subscription_sync_type parameter:

`CDB_DONE_PRIORITY` This means that the application has acted on the subscription notification and CDB can continue to deliver further notifications.

`CDB_DONE_SOCKET` This means that we are done. But regardless of priority, CDB shall not send any further notifications to us on our socket that are related to the currently executing transaction.

`CDB_DONE_TRANSACTION` This means that CDB should not send any further notifications to any subscribers - including ourselves - related to the currently executing transaction.

`CDB_DONE_OPERATIONAL` This should be used when a subscription notification for operational data has been read. It is the only type that should be used in this case, since the operational data does not have transactions and the notifications do not have priorities.

When using two phase subscriptions and `cdb_read_subscription_socket2()` has returned the type as `CDB_SUB_PREPARE` or `CDB_SUB_ABORT` the only valid response is `CDB_DONE_PRIORITY`.

For configuration data, the transaction that generated the subscription notifications is pending until all notifications have been acknowledged. A read lock on CDB is in effect while notifications are being delivered, preventing writes until delivery is complete.

For operational data, the writer that generated the subscription notifications is not directly affected, but the "subscription lock" remains in effect until all notifications have been acknowledged - thus subsequent attempts to obtain a "global" subscription lock, or a subscription lock using CDB_LOCK_PARTIAL for a non-disjuct subtree, will fail or block while notifications are being delivered (see `cdb_start_session2()` above). Write operations that don't attempt to obtain the subscription lock will proceed independent of the delivery of subscription notifications.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_get_modifications_cli`**`(int` *`sock`*`, int` *`subid`*`, int` *`flags`*`, char **`*`str`*`);`

The `cdb_get_modifications_cli()` function can be called after reception of a subscription notification to retrieve all the changes that caused the subscription notification as a string in Cisco CLI format. The socket *`s`* is the subscription socket, the subscription id must also be provided.

The CLI string is malloc(3)ed by the library, and the caller must free the memory using free(3) when it is not needed any longer.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_sub_progress`**`(int` *`sock`*`, const char *`*`fmt`*`, ...);`

After receiving a subscription notification (using `cdb_read_subscription_socket()`) but before acknowledging it (or aborting, in the case of prepare subscriptions), it is possible to send progress reports

back to ConfD using the `cdb_sub_progress()` function. The socket *`sock`* must be the subscription socket, and it is allowed to call the function more than once to display more than one message. It is also possible to use this function in the diff-iterate callback function. A newline at the end of the string isn't necessary.

Depending on which north-bound interface that triggered the transaction, the string passed may be reported by that interface. Currently this is only presented in the CLI when the operator requests detailed reporting using the **commit | details** command.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_sub_abort_trans`**`(int` *`sock`*`, enum confd_errcode` *`code`*`, u_int32_t` *`apptag_ns`*`, u_int32_t` *`apptag_tag`*`, const char *`*`fmt`*`, ...);`

This function is to be called instead of `cdb_sync_subscription_socket()` when the subscriber wishes to abort the current transaction. It is only valid to call after `cdb_read_subscription_socket2()` has returned with type set to `CDB_SUB_PREPARE`. The arguments after sock are the same as to `confd_X_seterr_extended()` and give the caller a way of indicating the reason for the failure. Details can be found in the EXTENDED ERROR REPORTING section in the confd_lib_lib(3) manpage.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_sub_abort_trans_info`**`(int` *`sock`*`, enum confd_errcode` *`code`*`, u_int32_t` *`apptag_ns`*`, u_int32_t` *`apptag_tag`*`, const confd_tag_value_t *`*`error_info`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

This function does the same as `cdb_sub_abort_trans()`, and additionally gives the possibility to provide contents for the NETCONF <error-info> element. See the EXTENDED ERROR REPORTING section in the confd_lib_lib(3) manpage.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`cdb_get_user_session`**`(int` *`sock`*`);`

Returns the user session id for the transaction that triggered the current subscription notification. This function uses a subscription socket, and can only be called when a subscription notification for configuration data has been received on that socket, before `cdb_sync_subscription_socket()` has been called. Additionally, it is not possible to call this function from the `iter()` function passed to `cdb_diff_iterate()`. To retrieve full information about the user session, use `maapi_get_user_session()` (see confd_lib_maapi(3)).

Note

Note: When the ConfD High Availability functionality is used, the user session information is not available on slave nodes.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADSTATE, CONFD_ERR_NOEXISTS

`int` **`cdb_get_transaction_handle`**`(int` *`sock`*`);`

Returns the transaction handle for the transaction that triggered the current subscription notification. This function uses a subscription socket, and can only be called when a subscription notification for configuration data has been received on that socket, before `cdb_sync_subscription_socket()` has

been called. Additionally, it is not possible to call this function from the `iter()` function passed to `cdb_diff_iterate()`.

Note

A CDB client is not expected to access the ConfD transaction store directly - this function should only be used for logging or debugging purposes.

Note

When the ConfD High Availability functionality is used, the transaction information is not available on slave nodes.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADSTATE, CONFD_ERR_NOEXISTS

```
int cdb_get(int sock, confd_value_t *v, const char *fmt, ...);
```

This reads a a value from the path in *`fmt`* and writes the result into the result parameter confd_value_t. The path must lead to a leaf element in the XML data tree. Note that for the C_BUF, C_BINARY, C_LIST, C_OBJECTREF, C_OID, C_QNAME, and C_HEXSTR confd_value_t types the buffer(s) pointed to are allocated using malloc(3), it is up to the user of this interface to free them using `confd_free_value()`.

Errors: CONFD_ERR_NOEXISTS, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE

All the type safe versions of `cdb_get()` described below, as well as `cdb_vget()`, also have the same possible Errors. When the type of the read value is wrong, `confd_errno` is set to CONFD_ERR_BADTYPE and the function returns CONFD_ERR. The YANG type is given in the descriptions below.

```
int cdb_get_int8(int sock, int8_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read int8 values.

```
int cdb_get_int16(int sock, int16_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read int16 values.

```
int cdb_get_int32(int sock, int32_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read int32 values.

```
int cdb_get_int64(int sock, int64_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read int64 values.

```
int cdb_get_u_int8(int sock, u_int8_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read uint8 values.

```
int cdb_get_u_int16(int sock, u_int16_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read uint16 values.

```
int cdb_get_u_int32(int sock, u_int32_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read uint32 values.

```
int cdb_get_u_int64(int sock, u_int64_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read uint64 values.

```
int cdb_get_bit32(int sock, u_int32_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read bits values where the highest assigned bit position for the type is 31.

```
int cdb_get_bit64(int sock, u_int64_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read bits values where the highest assigned bit position for the type is above 31.

```
int cdb_get_ipv4(int sock, struct in_addr *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read inet:ipv4-address values.

```
int cdb_get_ipv6(int sock, struct in6_addr *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read inet:ipv6-address values.

```
int cdb_get_double(int sock, double *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read xs:float and xs:double values.

```
int cdb_get_bool(int sock, int *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read boolean values.

```
int cdb_get_datetime(int sock, struct confd_datetime *rval, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read date-and-time values.

```
int cdb_get_date(int sock, struct confd_date *rval, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read xs:date values.

```
int cdb_get_time(int sock, struct confd_time *rval, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read xs:time values.

```
int cdb_get_duration(int sock, struct confd_duration *rval, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read xs:duration values.

```
int cdb_get_enum_value(int sock, int32_t *rval, const char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read enumeration values. If we have:

```
typedef unboundedType {
```

```
  type enumeration {
    enum unbounded;
    enum infinity;
  }
}
```

The two enumeration values `unbounded` and `infinity` will occur as two #define integers in the .h file which is generated from the YANG module. Thus this function `cdb_get_enum_value()` populates an unsigned integer pointer.

`int` **`cdb_get_objectref`**`(int` *`sock`*`, confd_hkeypath_t **`*`rval`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read instance-identifier values. Upon successful return *`rval`* is pointing to an allocated confd_hkeypath_t. It is up to the user of this function to free the hkeypath using `confd_free_hkeypath()` when it is not needed any longer.

`int` **`cdb_get_oid`**`(int` *`sock`*`, struct confd_snmp_oid **`*`rval`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read object-identifier values. Upon successful return *`rval`* is pointing to an allocated struct confd_snmp_oid. It is up to the user of this function to free the struct using free(3) when it is not needed any longer.

`int` **`cdb_get_buf`**`(int` *`sock`*`, unsigned char **`*`rval`*`, int *`*`bufsiz`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read string values. Upon successful return *`rval`* is pointing to a buffer of size *`bufsiz`*. It is up to the user of this function to free the buffer using free(3) when it is not needed any longer.

`int` **`cdb_get_buf2`**`(int` *`sock`*`, unsigned char *`*`rval`*`, int *`*`n`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read string values. If the buffer returned by `cdb_get()` fits into *`*n`* bytes CONFD_OK is returned and the buffer is copied into *`*rval`*. Upon successful return *`*n`* is set to the number of bytes copied into *`*rval`*.

`int` **`cdb_get_str`**`(int` *`sock`*`, char *`*`rval`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read string values. If the buffer returned by `cdb_get()` plus a terminating NUL fits into *`n`* bytes CONFD_OK is returned and the buffer is copied into *`*rval`* (as well as a terminating NUL character).

`int` **`cdb_get_binary`**`(int` *`sock`*`, unsigned char **`*`rval`*`, int *`*`bufsiz`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()`, as `cdb_get_buf()` but for binary values. Upon successful return *`rval`* is pointing to a buffer of size *`bufsiz`*. It is up to the user of this function to free the buffer using free(3) when it is not needed any longer.

`int` **`cdb_get_qname`**`(int` *`sock`*`, unsigned char **`*`prefix`*`, int *`*`prefixsz`*`, unsigned char **`*`name`*`, int *`*`namesz`*`, const char *`*`fmt`*`, ...);`

Type safe variant of `cdb_get()` which is used to read xs:QName values. Note that *`prefixsz`* can be zero (in which case *`*prefix`* will be set to NULL). The space for prefix and name is allocated using `malloc()`, it is up to the user of this function to free them when no longer in use.

```
int cdb_get_list(int sock, confd_value_t **values, int *n, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read values of a YANG `leaf-list`. The function will `malloc()` an array of confd_value_t elements for the list, and return a pointer to the array via the `**values` parameter and the length of the array via the `*n` parameter. The caller must free the memory for the values (see `cdb_get()`) and the array itself. An example that reads and prints the elements of a list of strings:

```
confd_value_t *values;
int i, n;

cdb_get_list(sock, &values, &n, "/system/cards");
for (i = 0; i < n; i++) {
    printf("card %d: %s\n", i, CONFD_GET_BUFPTR(&values[i]));
    confd_free_value(&values[i]);
}
free(values);
```

```
int cdb_get_ipv4prefix(int sock, struct confd_ipv4_prefix *rval, const
char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read inet:ipv4-prefix values.

```
int cdb_get_ipv6prefix(int sock, struct confd_ipv6_prefix *rval, const
char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read inet:ipv6-prefix values.

```
int cdb_get_decimal64(int sock, struct confd_decimal64 *rval, const char
*fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read decimal64 values.

```
int cdb_get_identityref(int sock, struct confd_identityref *rval, const
char *fmt, ...);
```

Type safe variant of `cdb_get()` which is used to read identityref values.

```
int cdb_vget(int sock, confd_value_t *v, const char *fmt, va_list args);
```

This function does the same as `cdb_get()`, but takes a single va_list argument instead of a variable number of arguments - i.e. similar to `vprintf()`. Corresponding va_list variants exist for all the functions that take a path as a variable number of arguments.

```
int cdb_get_object(int sock, confd_value_t *values, int n, const char
*fmt, ...);
```

In some cases it can be motivated to read multiple values in one request - this will be more efficient since it only incurs a single round trip to ConfD, but usage is a bit more complex. This function reads at most `n` values from the container or list entry specified by the path, and places them in the `values` array, which is provided by the caller. The array is populated according to the specification of the *Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page.

When reading from a container or list entry with mixed configuration and operational data (i.e. a config container or list entry that has some number of operational elements), some elements will have the "wrong" type - i.e. operational data in a session for CDB_RUNNING/CDB_STARTUP, or config data in a session

for CDB_OPERATIONAL. Leaf elements of the "wrong" type will have a "value" of C_NOEXISTS in the array, while static or (existing) optional sub-container elements will have C_XMLTAG in all cases. Sub-containers or leafs provided by external data providers will always be represented with C_NOEXISTS, whether config or not.

On success, the function returns the actual number of elements in the container or list entry. I.e. if the return value is bigger than *n*, only the values for the first *n* elements are in the array, and the remaining values have been discarded. Note that given the specification of the array contents, there is always a fixed upper bound on the number of actual elements, and if there are no presence sub-containers, the number is constant.

As an example, with the YANG fragment in the PATHS section above, this code could be used to read the values for interface "eth0" on host "buzz":

```
char *path = "/hosts/host{buzz}/interfaces/interface{%s}";
confd_value_t v[4];
struct in_addr ip, mask;
int enabled;

cdb_get_object(sock, v, 4, path, "eth0");
/* v[0] is interface name, already known
   - must be freed since it's a C_BUF   */
confd_free_value(&v[0]);
ip = CONFD_GET_IPV4(&v[1]);
mask = CONFD_GET_IPV4(&v[2]);
enabled = CONFD_GET_BOOL(&v[3]);
```

In this simple example, we assumed that the application was aware of the details of the data model, specifically that a confd_value_t array of length 4 would be sufficient for the values we wanted to retrieve, and at which positions in the array those values could be found. If we make use of schema information loaded from the ConfD daemon into the library (see confd_types(3)), we can avoid "hardwiring" these details. The following, more complex, example does the same as the above, but using only the names (in the form of #defines from the header file generated by **confdc --emit-h**) of the relevant leafs:

```
char *path = "/hosts/host{buzz}/interfaces/interface{%s}";
struct confd_cs_node *object = confd_cs_node_cd(NULL, path);
struct confd_cs_node *cur;
int n = confd_max_object_size(object);
int i;
confd_value_t v[n];
struct in_addr ip, mask;
int enabled;

cdb_get_object(sock, v, n, path, "eth0");
for (cur = object->children, i = 0;
     cur != NULL;
     cur = confd_next_object_node(object, cur, &v[i]), i++) {
    switch (cur->tag) {
    case hst_ip:
        ip = CONFD_GET_IPV4(&v[i]);
        break;
    case hst_mask:
        mask = CONFD_GET_IPV4(&v[i]);
        break;
    case hst_enabled:
        enabled = CONFD_GET_BOOL(&v[i]);
        break;
    }
```

```
    /* always free - it is a no-op if not needed */
    confd_free_value(&v[i]);
}
```

See confd_lib_lib(3) for the specification of the `confd_max_object_size()` and `confd_next_object_node()` functions. Also worth noting is that the return value from `confd_max_object_size()` is a constant for a given node in a given data model - thus we could optimize the above by calling `confd_max_object_size()` only at the first invocation of `cdb_get_object()` for a given node, making use of the `opaque` element of struct confd_cs_node to store the value:

```
char *path = "/hosts/host{buzz}/interfaces/interface{%s}";
struct confd_cs_node *object = confd_cs_node_cd(NULL, path);
int n;
struct in_addr ip, mask;
int enabled;

if (object->opaque == NULL) {
    n = confd_max_object_size(object);
    object->opaque = (void *)n;
} else {
    n = (int)object->opaque;
}

{
    struct confd_cs_node *cur;
    confd_value_t v[n];
    int i;

    cdb_get_object(sock, v, n, path, "eth0");
    for (cur = object->children, i = 0;
         cur != NULL;
         cur = confd_next_object_node(object, cur, &v[i]), i++) {
        ...
    }
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

`int` **`cdb_get_objects`**`(int` *`sock`*`, confd_value_t *`*`values`*`, int` *`n`*`, int` *`ix`*`, int` *`nobj`*`, const char *`*`fmt`*`, ...);`

Similar to `cdb_get_object()`, but reads multiple entries of a list based on the "instance integer" otherwise given within square brackets in the path - here the path must specify the list without the instance integer. At most *`n`* values from each of *`nobj`* entries, starting at entry *`ix`*, are read and placed in the *`values`* array.

The array must be at least `n * nobj` elements long, and the values for list entry `ix + i` start at element `array[i * n]` (i.e. `ix` starts at `array[0]`, `ix+1` at `array[n]`, and so on). On success, the highest actual number of values in any of the list entries read is returned. An error (CONFD_ERR_NOEXISTS) will be returned if we attempt to read more entries than actually exist (i.e. if `ix + nobj - 1` is outside the range of actually existing list entries). Example - read the data for all interfaces on the host "buzz" (assuming that we have memory enough for that):

```
char *path = "/hosts/host{buzz}/interfaces/interface";
int n;

n = cdb_num_instances(sock, path);
```

```
{
    confd_value_t v[n*4];
    char name[n][64];
    struct in_addr ip[n], mask[n];
    int enabled[n];
    int i;

    cdb_get_objects(sock, v, 4, 0, n, path);
    for (i = 0; i < n*4; i += 4) {
        confd_pp_value(&name[i][0], 64, &v[i]);
        /* value must be freed since it's a C_BUF */
        confd_free_value(&v[i]);
        ip[i] = CONFD_GET_IPV4(&v[i+1]);
        mask[i] = CONFD_GET_IPV4(&v[i+2]);
        enabled[i] = CONFD_GET_BOOL(&v[i+3]);
    }

    /* configure interfaces... */
}
```

This simple example can of course be enhanced to use loaded schema information in a similar manner as for `cdb_get_object()` above.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

`int` **`cdb_get_values`**`(int` *`sock`*`, confd_tag_value_t *`*`values`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

Read an arbitrary set of sub-elements of a container or list entry. The *`values`* array must be pre-populated with *`n`* values based on the specification of the *Tagged Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page, where the confd_value_t value element is given as follows:

- C_NOEXISTS means that the value should be read from CDB and stored in the array.
- C_PTR also means that the value should be read from CDB, but instead gives the expected type and a pointer to the type-specific variable where the value should be stored. Thus this gives a functionality similar to the type safe versions of `cdb_get()`.
- C_XMLBEGIN and C_XMLEND are used as per the specification.
- Key values to select list entries can be given with their values.
- As a special case, the "instance integer" can be used to select a list entry by using C_CDBBEGIN instead of C_XMLBEGIN (and no key values).

Note

When we use C_PTR, we need to take special care to free any allocated memory. When we use C_NOEXISTS and the value is stored in the array, we can just use `confd_free_value()` regardless of the type, since the confd_value_t has the type information. But with C_PTR, only the actual value is stored in the pointed-to variable, just as for `cdb_get_buf()`, `cdb_get_binary()`, etc, and we need to free the memory specifically allocated for the types listed in the description of `cdb_get()` above. See the corresponding `cdb_get_xxx()` functions for the details of how to do this.

All elements have the same position in the array after the call, in order to simplify extraction of the values - this means that optional elements that were requested but didn't exist will have C_NOEXISTS rather than

being omitted from the array. However requesting a list entry that doesn't exist, or requesting non-CDB data, or operational vs config data, is an error. Note that when using C_PTR, the only indication of a non-existing value is that the destination variable has not been modified - it's up to the application to set it to some "impossible" value before the call when optional leafs are read.

In this rather complex example we first read only the "name" and "enabled" values for all interfaces, and then read "ip" and "mask" for those that were enabled - a total of two requests. Note that since the "interface" list begin/end elements are in the array, the path must not include the "interface" component. When reading values from a single container, it is generally simpler to have the container component (and keys or instance integer) in the path instead.

```
char *path = "/hosts/host{buzz}/interfaces";
int n = cdb_num_instances(sock, "%s/interface", path);
{
  /* when reading ip/mask, we need 5 elements per interface:
     begin + name (key) + ip + mask + end                    */
  confd_tag_value_t tv[n*5];
  char name[n][64];
  struct in_addr ip[n], mask[n];
  int i, j;
  int n_if;

  /* read name and enabled for all interfaces */
  j = 0;
  for (i = 0; i < n; i++) {
    CONFD_SET_TAG_CDBBEGIN(&tv[j], hst_interface, hst__ns, i); j++;
    CONFD_SET_TAG_NOEXISTS(&tv[j], hst_name);                  j++;
    CONFD_SET_TAG_NOEXISTS(&tv[j], hst_enabled);               j++;
    CONFD_SET_TAG_XMLEND(&tv[j], hst_interface, hst__ns);      j++;
  }
  cdb_get_values(sock, tv, j, path);

  /* extract name for enabled interfaces */
  j = 0;
  for (i = 0; i < n*4; i += 4) {
    int enabled = CONFD_GET_BOOL(CONFD_GET_TAG_VALUE(&tv[i+2]));
    confd_value_t *v = CONFD_GET_TAG_VALUE(&tv[i+1]);
    if (enabled) {
      confd_pp_value(&name[j][0], 64, v);
      j++;
    }
    /* name must be freed regardless since it's a C_BUF */
    confd_free_value(v);
  }
  n_if = j;

  /* read ip and mask for enabled interfaces by key value (name) */
  j = 0;
  for (i = 0; i < n_if; i++) {
    CONFD_SET_TAG_XMLBEGIN(&tv[j], hst_interface, hst__ns);    j++;
    CONFD_SET_TAG_STR(&tv[j], hst_name, &name[i][0]);          j++;
    CONFD_SET_TAG_PTR(&tv[j], hst_ip, C_IPV4, &ip[i]);         j++;
    CONFD_SET_TAG_PTR(&tv[j], hst_mask, C_IPV4, &mask[i]);     j++;
    CONFD_SET_TAG_XMLEND(&tv[j], hst_interface, hst__ns);      j++;
  }
  cdb_get_values(sock, tv, j, path);

  for (i = 0; i < n_if; i++) {
    /* configure interface i with ip[i] and mask[i]... */
```

```
  }
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOEXISTS

`int` **`cdb_get_case`**`(int` *`sock`*`, const char *`*`choice`*`, confd_value_t *`*`rcase`*`, const char *`*`fmt`*`, ...);`

When we use the YANG `choice` statement in the data model, this function can be used to find the currently selected `case`, avoiding useless `cdb_get()` etc requests for elements that belong to other cases. The *`fmt, ...`* arguments give the path to the container or list entry where the choice is defined, and *`choice`* is the name of the choice. The case value is returned to the confd_value_t that *`rcase`* points to, as type C_XMLTAG - i.e. we can use the `CONFD_GET_XMLTAG()` macro to retrieve the hashed tag value. If no case is currently selected (i.e. for an optional choice that doesn't have a default case), the function will fail with CONFD_ERR_NOEXISTS.

If we have "nested" choices, i.e. multiple levels of `choice` statements without intervening `container` or `list` statements in the data model, the *`choice`* argument must give a '/'-separated path with alternating choice and case names, from the data node given by the *`fmt, ...`* arguments to the specific choice that the request pertains to.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

OPERATIONAL DATA

It is possible for an application to store operational data (i.e. status and statistical information) in CDB, instead of providing it on demand via the callback interfaces described in the confd_lib_dp(3) manual page. The operational database has no transactions and normally avoids the use of locks in order to provide light-weight access methods, however when the multi-value API functions below are used, all updates requested by a given function call are carried out atomically. Read about how to specify the storage of operational data in CDB via the `tailf:cdb-oper` extension in the tailf_yang_extensions(5) manual page.

To establish a session for operational data, the application needs to use `cdb_connect()` with CDB_DATA_SOCKET and `cdb_start_session()` with CDB_OPERATIONAL. After this, all the read and access functions above are available for use with operational data, and additionally the write functions described below. Configuration data can not be accessed in a session for operational data, nor vice versa - however it is possible to have both types of sessions active simultaneously on two different sockets, or to alternate the use of one socket via `cdb_end_session()`. The write functions can never be used in a session for configuration data.

Note

In order to trigger subscriptions on operational data, we must obtain a subscription lock via the use of `cdb_start_session2()` instead of `cdb_start_session()`, see above.

In YANG it is possible to define a list of operational data without any keys. For this type of list, we use a single "pseudo" key which is always of type C_INT64 - see the Operational Data chapter in the User Guide. This key isn't visible in the northbound agent interfaces, but is used in the functions described here just as if it was a "normal" key.

`int` **`cdb_set_elem`**`(int` *`sock`*`, confd_value_t *`*`val`*`, const char *`*`fmt`*`, ...);`

`int` **`cdb_set_elem2`**`(int` *`sock`*`, const char *`*`strval`*`, const char *`*`fmt`*`, ...);`

There are two different functions to set the value of a single leaf. The first takes the value from a confd_value_t struct, the second takes the string representation of the value.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE

`int` **`cdb_vset_elem`**`(int` *`sock`*`, confd_value_t *`*`val`*`, const char *`*`fmt`*`, va_list` *`args`*`);`

This function does the same as `cdb_set_elem()`, but takes a single va_list argument instead of a variable number of arguments - i.e. similar to `vprintf()`. Corresponding va_list variants exist for all the functions that take a path as a variable number of arguments.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE

`int` **`cdb_create`**`(int` *`sock`*`, const char *`*`fmt`*`, ...);`

Create a new list entry, presence container, or leaf of type `empty`. Note that for list entries and containers, sub-elements will not exist until created or set via some of the other functions, thus doing implicit create via `cdb_set_object()` or `cdb_set_values()` may be preferred in this case.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOTCREATABLE, CONFD_ERR_ALREADY_EXISTS

`int` **`cdb_delete`**`(int` *`sock`*`, const char *`*`fmt`*`, ...);`

Delete a list entry, presence container, or leaf of type `empty`, and all its child elements (if any).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOTDELETABLE, CONFD_ERR_NOEXISTS

`int` **`cdb_set_object`**`(int` *`sock`*`, const confd_value_t *`*`values`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

Set all elements corresponding to the complete contents of a container or list entry, except for sub-lists. The *`values`* array must be populated with *`n`* values according to the specification of the *Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page.

If the container or list entry itself, or any sub-elements that are specified as existing, do not exist before this call, they will be created, otherwise the existing values will be updated. Non-mandatory leafs and presence containers that are specified as not existing in the array, i.e. with value C_NOEXISTS, will be deleted if they existed before the call.

When writing to a container with mixed configuration and operational data (i.e. a config container or list entry that has some number of operational elements), all config leaf elements must be specified as C_NOEXISTS in the corresponding array elements, while config sub-container elements are specified with C_XMLTAG just as for operational data.

For a list entry, since the key elements must be present in the array, it is not required that the key values are included in the path given by *`fmt`*. If the key values *are* included in the path, the values of the key elements in the array are ignored.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE

`int` **`cdb_set_values`**`(int sock, const confd_tag_value_t *values, int n, const char *fmt, ...);`

Set arbitrary sub-elements of a container or list entry. The *`values`* array must be populated with *`n`* values according to the specification of the *Tagged Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page.

If the container or list entry itself, or any sub-elements that are specified as existing, do not exist before this call, they will be created, otherwise the existing values will be updated. Both mandatory and optional elements may be omitted from the array, and all omitted elements are left unchanged. To actually delete a non-mandatory leaf or presence container as described for `cdb_set_object()`, it may (as an extension of the format) be specified as C_NOEXISTS instead of being omitted.

For a list entry, the key values can be specified either in the path or via key elements in the array - if the values are in the path, the key elements can be omitted from the array. For sub-lists present in the array, the key elements must of course always also be present though, immediately following the C_XMLBEGIN element and in the order defined by the data model. It is also possible to delete a list entry by using a C_XMLBEGINDEL element, followed by the keys in data model order, followed by a C_XMLEND element.

For a list without keys (see above), the "pseudo" key may (or in some cases must) be present in the array, but of course there is no tag value for it, since it isn't present in the data model. In this case we must use a tag value of 0, i.e. it can be set with code like:

```
confd_tag_value_t tv[7];

CONFD_SET_TAG_INT64(&tv[1], 0, 42);
```

The same method is used when reading data from such a list with the `cdb_get_values()` function described above.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE

`int` **`cdb_set_case`**`(int sock, const char *choice, const char *scase, const char *fmt, ...);`

When we use the YANG `choice` statement in the data model, this function can be used to select the current `case`. When configuration data is modified by northbound agents, the current case is implicitly selected (and elements for other cases potentially deleted) by the setting of elements in a choice. For operational data in CDB however, this is under direct control of the application, which needs to explicitly set the current case. Setting the case will also automatically delete elements belonging to other cases, but it is up to the application to not set any elements in the "wrong" case.

The *`fmt, ...`* arguments give the path to the container or list entry where the choice is defined, and *`choice`* and *`scase`* are the choice and case names. For an optional choice, it is possible to have no case at all selected. To indicate that the previously selected case should be deleted without selecting another case, we can pass NULL for the *`scase`* argument.

If we have "nested" choices, i.e. multiple levels of `choice` statements without intervening `container` or `list` statements in the data model, the *`choice`* argument must give a '/'-separated path with alternating choice and case names, from the data node given by the *`fmt, ...`* arguments to the specific choice that the request pertains to.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOTDELETABLE

```
int cdb_load_file(int sock, const char *filename, int flags);
```

Load operational data from *filename* into CDB operational. The file must be in xml format, and *sock* must be connected to CDB operational (i.e. `cdb_start_session()` or `cdb_start_session2()` must have been called with CDB_OPERATIONAL). If the file contains config data, or operational data not residing in CDB, that data will be silently ignored. If the name of the file ends in .gz (or .Z) then the file is assumed to be gzipped, and will be uncompressed as it is loaded.

Note

If you use a relative pathname for *filename*, it is taken as relative to the working directory of the ConfD daemon, i.e. the directory where the daemon was started.

Note that there are no transactions in CDB operational, so there will not be any validation or transactional commit of the file. However the file will be completely parsed before CDB tries to set the values, with the result that any errors in the file will abort the operation without changing anything in CDB operational.

The flags parameter is currently not used, and should be set to 0.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_BADPATH, CONFD_ERR_EXTERNAL

```
int cdb_load_str(int sock, const char *xml_str, int flags);
```

Load operational data from the string *xml_str* into CDB operational. I.e. instead of having the xml data read from a file as for `cdb_load_file()`, it is passed as a string to the function. Besides this, the function works the same as `cdb_load_file()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_EXTERNAL

SEE ALSO

confd_lib(3) - Confd lib

confd_types(3) - ConfD C data types

The ConfD User Guide

Name

confd_lib_dp — callback library for connecting data providers to ConfD

Synopsis

#include <confd_lib.h> #include <confd_dp.h>

```
struct confd_daemon_ctx *confd_init_daemon(const char *name);

int confd_set_daemon_flags(struct confd_daemon_ctx *dx, int flags);

void confd_release_daemon(struct confd_daemon_ctx *dx);

int confd_connect(struct confd_daemon_ctx *dx, int sock, enum
confd_sock_type type, const struct sockaddr *srv, int addrsz);

int confd_register_trans_cb(struct confd_daemon_ctx *dx, const struct
confd_trans_cbs *trans);

int confd_register_db_cb(struct confd_daemon_ctx *dx, const struct
confd_db_cbs *dbcbs);

int confd_register_range_data_cb(struct confd_daemon_ctx *dx, con-
st struct confd_data_cbs *data, const confd_value_t *lower, const
confd_value_t *upper, int numkeys, const char *fmt, ...);

int confd_register_data_cb(struct confd_daemon_ctx *dx, const struct
confd_data_cbs *data);

int confd_register_usess_cb(struct confd_daemon_ctx *dx, const struct
confd_usess_cbs *ucb);

int ncs_register_service_cb(struct confd_daemon_ctx *dx, const struct
ncs_service_cbs *scb);

int confd_register_done(struct confd_daemon_ctx *dx);

int confd_fd_ready(struct confd_daemon_ctx *dx, int fd);

void confd_trans_set_fd(struct confd_trans_ctx *tctx, int sock);

int confd_data_reply_value(struct confd_trans_ctx *tctx, const
confd_value_t *v);

int confd_data_reply_value_array(struct confd_trans_ctx *tctx, const
confd_value_t *vs, int n);

int confd_data_reply_tag_value_array(struct confd_trans_ctx *tctx, con-
st confd_tag_value_t *tvs, int n);

int confd_data_reply_next_key(struct confd_trans_ctx *tctx, const
confd_value_t *v, int num_vals_in_key, long next);

int confd_data_reply_not_found(struct confd_trans_ctx *tctx);

int confd_data_reply_found(struct confd_trans_ctx *tctx);
```

int **confd_data_reply_next_object_array**(struct confd_trans_ctx *tctx, const confd_value_t *v, int n, long next);

int **confd_data_reply_next_object_tag_value_array**(struct confd_trans_ctx *tctx, const confd_tag_value_t *tv, int n, long next);

int **confd_data_reply_next_object_arrays**(struct confd_trans_ctx *tctx, const struct confd_next_object *obj, int nobj, int timeout_millisecs);

int **confd_data_reply_next_object_tag_value_arrays**(struct confd_trans_ctx *tctx, const struct confd_tag_next_object *tobj, int nobj, int timeout_millisecs);

int **confd_data_reply_attrs**(struct confd_trans_ctx *tctx, const confd_attr_value_t *attrs, int num_attrs);

int **ncs_service_reply_proplist**(struct confd_trans_ctx *tctx, const struct ncs_name_value *proplist, int num_props);

int **confd_delayed_reply_ok**(struct confd_trans_ctx *tctx);

int **confd_delayed_reply_error**(struct confd_trans_ctx *tctx, const char *errstr);

int **confd_data_set_timeout**(struct confd_trans_ctx *tctx, int timeout_secs);

void **confd_trans_seterr**(struct confd_trans_ctx *tctx, const char *fmt, ...);

void **confd_trans_seterr_extended**(struct confd_trans_ctx *tctx, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, const char *fmt, ...);

int **confd_trans_seterr_extended_info**(struct confd_trans_ctx *tctx, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, confd_tag_value_t *error_info, int n, const char *fmt, ...);

void **confd_db_seterr**(struct confd_db_ctx *dbx, const char *fmt, ...);

void **confd_db_seterr_extended**(struct confd_db_ctx *dbx, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, const char *fmt, ...);

int **confd_db_seterr_extended_info**(struct confd_db_ctx *dbx, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, confd_tag_value_t *error_info, int n, const char *fmt, ...);

int **confd_db_set_timeout**(struct confd_db_ctx *dbx, int timeout_secs);

int **confd_aaa_reload**(const struct confd_trans_ctx *tctx);

int **confd_install_crypto_keys**(struct confd_daemon_ctx* dtx);

void **confd_register_trans_validate_cb**(struct confd_daemon_ctx *dx, const struct confd_trans_validate_cbs *vcbs);

int **confd_register_valpoint_cb**(struct confd_daemon_ctx *dx, const struct confd_valpoint_cb *vcb);

int **confd_register_range_valpoint_cb**(struct confd_daemon_ctx *dx, struct confd_valpoint_cb *vcb, const confd_value_t *lower, const confd_value_t *upper, int numkeys, const char *fmt, ...);

int **confd_delayed_reply_validation_warn**(struct confd_trans_ctx *tctx);

int **confd_register_action_cbs**(struct confd_daemon_ctx *dx, const struct confd_action_cbs *acb);

int **confd_register_range_action_cbs**(struct confd_daemon_ctx *dx, const struct confd_action_cbs *acb, const confd_value_t *lower, const confd_value_t *upper, int numkeys, const char *fmt, ...);

void **confd_action_set_fd**(struct confd_user_info *uinfo, int sock);

void **confd_action_seterr**(struct confd_user_info *uinfo, const char *fmt, ...);

void **confd_action_seterr_extended**(struct confd_user_info *uinfo, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, const char *fmt, ...);

int **confd_action_seterr_extended_info**(struct confd_user_info *uinfo, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, confd_tag_value_t *error_info, int n, const char *fmt, ...);

int **confd_action_reply_values**(struct confd_user_info *uinfo, confd_tag_value_t *values, int nvalues);

int **confd_action_reply_command**(struct confd_user_info *uinfo, char **values, int nvalues);

int **confd_action_reply_rewrite**(struct confd_user_info *uinfo, char **values, int nvalues, char **unhides, int nunhides);

int **confd_action_reply_rewrite2**(struct confd_user_info *uinfo, char **values, int nvalues, char **unhides, int nunhides, struct confd_rewrite_select **selects, int nselects);

int **confd_action_reply_completion**(struct confd_user_info *uinfo, struct confd_completion_value *values, int nvalues);

int **confd_action_reply_range_enum**(struct confd_user_info *uinfo, char **values, int keysize, int nkeys);

int **confd_action_delayed_reply_ok**(struct confd_user_info *uinfo);

int **confd_action_delayed_reply_error**(struct confd_user_info *uinfo, const char *errstr);

int **confd_action_set_timeout**(struct confd_user_info *uinfo, int timeout_secs);

int **confd_register_notification_stream**(struct confd_daemon_ctx *dx, const struct confd_notification_stream_cbs *ncbs, struct confd_notification_ctx **nctx);

int **confd_notification_send**(struct confd_notification_ctx *_nctx_, struct confd_datetime *_time_, confd_tag_value_t *_values_, int _nvalues_);

int **confd_notification_replay_complete**(struct confd_notification_ctx *_nctx_);

int **confd_notification_replay_failed**(struct confd_notification_ctx *_nctx_);

int **confd_notification_reply_log_times**(struct confd_notification_ctx *_nctx_, struct confd_datetime *_creation_, struct confd_datetime *_aged_);

void **confd_notification_set_fd**(struct confd_notification_ctx *_nctx_, int _fd_);

void **confd_notification_set_snmp_src_addr**(struct confd_notification_ctx *_nctx_, const struct confd_ip *_src_addr_);

int **confd_notification_set_snmp_notify_name**(struct confd_notification_ctx *_nctx_, const char *_notify_name_);

void **confd_notification_seterr**(struct confd_notification_ctx *_nctx_, const char *_fmt_, ...);

void **confd_notification_seterr_extended**(struct confd_notification_ctx *_nctx_, enum confd_errcode _code_, u_int32_t _apptag_ns_, u_int32_t _apptag_tag_, const char *_fmt_, ...);

int **confd_notification_seterr_extended_info**(struct confd_notification_ctx *_nctx_, enum confd_errcode _code_, u_int32_t _apptag_ns_, u_int32_t _apptag_tag_, confd_tag_value_t *_error_info_, int _n_, const char *_fmt_, ...);

int **confd_register_snmp_notification**(struct confd_daemon_ctx *_dx_, int _fd_, const char *_notify_name_, const char *_ctx_name_, struct confd_notification_ctx **_nctx_);

int **confd_notification_send_snmp**(struct confd_notification_ctx *_nctx_, const char *_notification_, struct confd_snmp_varbind *_varbinds_, int _num_vars_);

int **confd_register_notification_snmp_inform_cb**(struct confd_daemon_ctx *_dx_, const struct confd_notification_snmp_inform_cbs *_cb_);

int **confd_notification_send_snmp_inform**(struct confd_notification_ctx *_nctx_, const char *_notification_, struct confd_snmp_varbind *_varbinds_, int _num_vars_, const char *_cb_id_, int _ref_);

int **confd_register_notification_sub_snmp_cb**(struct confd_daemon_ctx *_dx_, const struct confd_notification_sub_snmp_cb *_cb_);

int **confd_notification_flush**(struct confd_notification_ctx *_nctx_);

int **confd_register_auth_cb**(struct confd_daemon_ctx *_dx_, const struct confd_auth_cb *_acb_);

void **confd_auth_seterr**(struct confd_auth_ctx *_actx_, const char *_fmt_, ...);

```
int confd_register_authorization_cb(struct confd_daemon_ctx *dx, const
struct confd_authorization_cbs *acb);

int confd_access_reply_result(struct confd_authorization_ctx *actx, int
result);

int confd_authorization_set_timeout(struct confd_authorization_ctx *ac-
tx, int timeout_secs);

int confd_register_error_cb(struct confd_daemon_ctx *dx, const struct
confd_error_cb *ecb);

void confd_error_seterr(struct confd_user_info *uinfo, const char
*fmt, ...);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to the ConfD Data Provider API. The purpose of this API is to provide callback hooks so that user-written data providers can provide data stored externally to ConfD. ConfD needs this information in order to drive its northbound agents.

The library is also used to populate items in the data model which are not data or configuration items, such as statistics items from the device.

The library consists of a number of API functions whose purpose is to install different callback functions at different points in the data model tree which is the representation of the device configuration. Read more about callpoints in tailf_yang_extensions(5). Read more about how to use the library in the User Guide chapters on Operational data and External data.

FUNCTIONS

```
struct confd_daemon_ctx *confd_init_daemon(const char *name);
```

Initializes a new daemon context or returns NULL on failure. For most of the library functions described here a daemon_ctx is required, so we must create a daemon context before we can use them. The daemon context contains a d_opaque pointer which can be used by the application to pass application specific data into the callback functions.

The *`name`* parameter is used in various debug printouts and and is also used to uniquely identify the daemon. The **confd --status** will use this name when indicating which callpoints are registered.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE

```
int confd_set_daemon_flags(struct confd_daemon_ctx *dx, int flags);
```

This function modifies the API behaviour according to the flags ORed into the *`flags`* argument. It should be called immediately after creating the daemon context with `confd_init_daemon()`. The following flags are available:

`CONFD_DAEMON_FLAG_STRINGSONLY`
: If this flag is used, the callback functions described below will only receive string values for all instances of confd_value_t (i.e. the type is always C_BUF). The callbacks must also give only string

values in their reply functions. This feature can be useful for proxy-type applications that are unaware of the types of all elements, i.e. data model agnostic.

`CONFD_DAEMON_FLAG_REG_REPLACE_DISCONNECT`
By default, if one daemon replaces a callpoint registration made by another daemon, this is only logged, and no action is taken towards the daemon that has "lost" its registration. This can be useful in some scenarios, e.g. it is possible to have an "initial default" daemon providing "null" data for many callpoints, until the actual data provider daemons have registered. If a daemon uses the `CONFD_DAEMON_FLAG_REG_REPLACE_DISCONNECT` flag, it will instead be disconnected from ConfD if any of its registrations are replaced by another daemon, and can take action as appropriate.

`CONFD_DAEMON_FLAG_NO_DEFAULTS`
This flag tells ConfD that the daemon does not store default values. By default, ConfD assumes that the daemon doesn't know about default values, and thus whenever default values come into effect, ConfD will issue `set_elem()` callbacks to set those values, even if they have not actually been set by the northbound agent. Similarly `set_case()` will be issued with the default case for choices that have one.

When the `CONFD_DAEMON_FLAG_NO_DEFAULTS` flag is set, ConfD will only issue `set_elem()` callbacks when values have been explicitly set, and `set_case()` when a case has been selected by explicitly setting an element in the case. Specifically:

- When a list entry or presence container is created, there will be no callbacks for descendant leafs with default value, or descendant choices with default case, unless values have been explicitly set.
- When a leaf with a default value is deleted, a `remove()` callback will be issued instead of a `set_elem()` with the default value.
- When the current case in a choice with default case is deleted without another case being selected, the `set_case()` callback will be invoked with the case value given as NULL instead of the default case.

Note

A daemon that has the `CONFD_DAEMON_FLAG_NO_DEFAULTS` flag set *must* reply to `get_elem()` and the other callbacks that request leaf values with a value of type C_DEFAULT, rather than the actual default value, when the default value for a leaf is in effect. It *must* also reply to `get_case()` with C_DEFAULT when the default case is in effect.

```
void confd_release_daemon(struct confd_daemon_ctx *dx);
```

Returns all memory that has been allocated by `confd_init_daemon()` and other functions for the daemon context. The control socket as well as all the worker sockets must be closed by the application (before or after `confd_release_daemon()` has been called).

```
int confd_connect(struct confd_daemon_ctx *dx, int sock, enum
confd_sock_type type, const struct sockaddr *srv, int addrsz);
```

Connects to the ConfD daemon. The *dx* parameter is a daemon context acquired through a call to `confd_init_daemon()`.

There are two different types of connected sockets between an external daemon and ConfD.

`CONTROL_SOCKET` The first socket that is connected must always be a control socket. All requests from ConfD to create new transactions will arrive on the control socket, but it is also used for a number of other requests that are expected to complete quickly - the general rule is that all callbacks that do not have a corresponding `init()`

callback are in fact control socket requests. There can only be one control socket for a given daemon context.

`WORKER_SOCKET` We must always create at least one worker socket. All transaction, data, validation, and action callbacks, except the `init()` callbacks, use a worker socket. It is possible for a daemon to have multiple worker sockets, and the `init()` callback (see e.g. `confd_register_trans_cb()`) must indicate which worker socket should be used for the subsequent requests. This makes it possible for an application to be multi-threaded, where different threads can be used for different transactions.

Returns CONFD_OK when successful or CONFD_ERR on connection error.

Note

All the callbacks that are invoked via these sockets are subject to timeouts configured in `confd.conf`, see confd.conf(5). The callbacks invoked via the control socket must generate a reply back to ConfD within the time configured for `/confdConfig/capi/newSessionTimeout`, the callbacks invoked via a worker socket within the time configured for `/confdConfig/capi/queryTimeout`. If either timeout is exceeded, the daemon will be considered dead, and ConfD will disconnect it by closing the control and worker sockets.

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

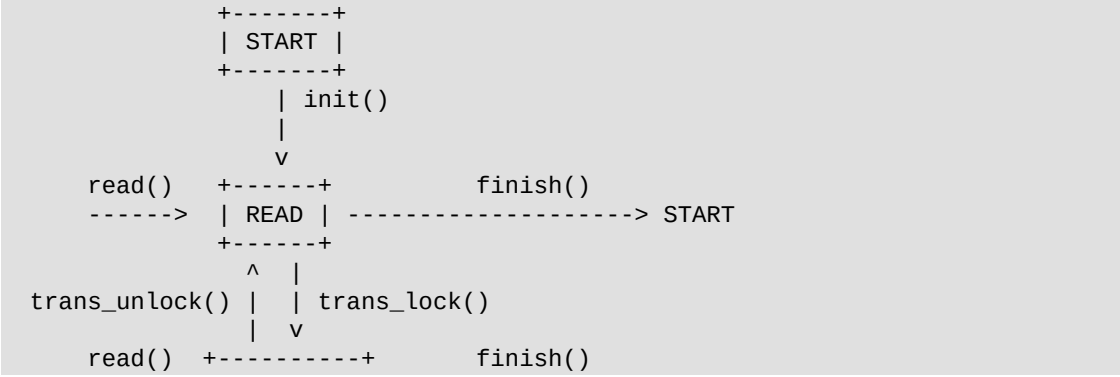
Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE

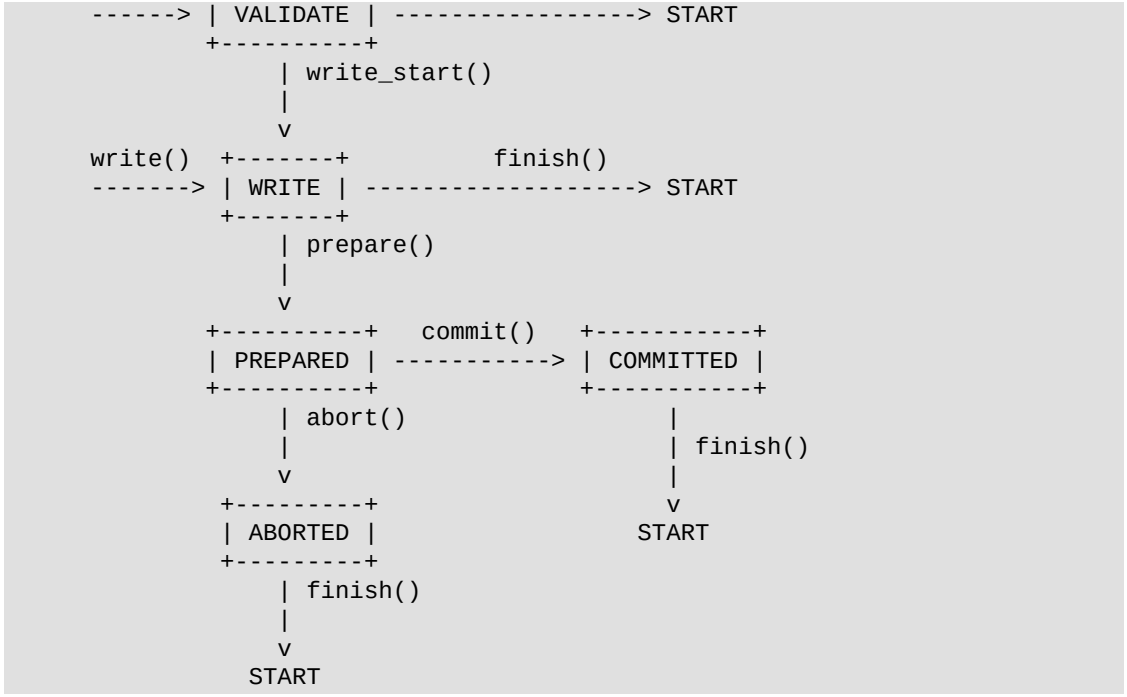
`int` **`confd_register_trans_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_trans_cbs *`*`trans`*`);`

This function registers transaction callback functions. A transaction is a ConfD concept. There may be multiple sources of data for the device configuration.

In order to orchestrate transactions with multiple sources of data, ConfD implements a two-phase commit protocol towards all data sources that participate in a transaction.

Each NETCONF operation will be an individual ConfD transaction. These transactions are typically very short lived. Transactions originating from the CLI or the Web UI have longer life. The ConfD transaction can be viewed as a conceptual state machine where the different phases of the transaction are different states and the invocations of the callback functions are state transitions. The following ASCII art depicts the state machine.





The struct confd_trans_cbs is defined as:

```
struct confd_trans_cbs {
    int (*init)(struct confd_trans_ctx *tctx);
    int (*trans_lock)(struct confd_trans_ctx *sctx);
    int (*trans_unlock)(struct confd_trans_ctx *sctx);
    int (*write_start)(struct confd_trans_ctx *sctx);
    int (*prepare)(struct confd_trans_ctx *tctx);
    int (*abort)(struct confd_trans_ctx *tctx);
    int (*commit)(struct confd_trans_ctx *tctx);
    int (*finish)(struct confd_trans_ctx *tctx);
    void (*interrupt)(struct confd_trans_ctx *tctx);
};
```

Transactions can be performed towards fours different kind of storages.

`CONFD_CANDIDATE` If the system has been configured so that the external database owns the candidate data share, we will have to execute candidate transactions here. Usually ConfD owns the candidate and in that case the external database will never see any CONFD_CANDIDATE transactions.

`CONFD_RUNNING` This is a transaction towards the actual running configuration of the device. All write operations in a CONFD_RUNNING transaction must be propagated to the individual subsystems that use this configuration data.

`CONFD_STARTUP` If the system has ben configured to support the NETCONF startup capability, this is a transaction towards the startup database.

`CONFD_OPERATIONAL` This value indicates a transaction towards writable operational data. This transaction is used only if there are non-config data marked as `tailf:writable true` in the YANG module.

Currently, these transaction are only started by the SNMP agent, and only when writable operational data is SET over SNMP.

Which type we have is indicated through the `confd_dbname` field in the confd_trans_ctx.

A transaction, regardless of whether it originates from the NETCONF agent, the CLI or the Web UI, has several distinct phases:

`init()`
: This callback must always be implemented. All other callbacks are optional. This means that if the callback is set to NULL, ConfD will treat it as an implicit CONFD_OK. `libconfd` will allocate a transaction context on behalf of the transaction and give this newly allocated structure as an argument to the `init()` callback. The structure is defined as:

```
struct confd_user_info {
    int af;                         /* AF_INET | AF_INET6 */
    union {
        struct in_addr v4;          /* address from where the */
        struct in6_addr v6;         /* user session originates */
    } ip;
    u_int16_t port;                 /* source port */
    char username[MAXUSERNAMELEN]; /* who is the user */
    int usid;                       /* user session id */
    char context[MAXCTXLEN];        /* cli | webui | netconf | */
                                    /* noaaa | any MAAPI string */
    enum confd_proto proto;         /* which protocol */
    struct confd_action_ctx actx;  /* used during action call */
    time_t logintime;
    enum confd_usess_lock_mode lmode;  /* the lock we have (only from */
                                       /* maapi_get_user_session())   */
    char snmp_v3_ctx[255];          /* SNMP context for SNMP sessions */
                                    /* empty string ("") for non-SNMP sessions */
    char clearpass[255];            /* if have the pass, it's here */
                                    /* only if confd internal ssh is used */
    int flags;                      /* CONFD_USESS_FLAG_... */
    void *u_opaque;                 /* Private User data */
    /* ConfD internal fields */
    char *errstr;                   /* for error formatting callback */
    int refc;
};
```

```
struct confd_trans_ctx {
    int fd;                      /* trans (worker) socket */
    int vfd;                     /* validation worker socket */
    struct confd_daemon_ctx *dx; /* our daemon ctx */
    enum confd_trans_mode mode;
    enum confd_dbname dbname;
    struct confd_user_info *uinfo;
    void *t_opaque;              /* Private User data (transaction) */
    void *v_opaque;              /* Private User data (validation) */
    struct confd_error error;    /* user settable via */
                                 /* confd_trans_seterr*() */
    struct confd_tr_item *accumulated;
    int thandle;                 /* transaction handle */
    void *cb_opaque;             /* private user data from */
                                 /* data callback registration */
    void *vcb_opaque;            /* private user data from */
                                 /* validation callback registration */
    int secondary_index;         /* if != 0: secondary index number */
                                 /* for list traversal */
    int validation_info;         /* CONFD_VALIDATION_FLAG_XXX */
    char *callpoint_opaque;      /* tailf:opaque for callpoint
                                    in data model */
```

```
    char *validate_opaque;       /* tailf:opaque for validation point
                                    in data model */
    union confd_request_data request_data; /* info from northbound agent */
    int hide_inactive;           /* if != 0: config data with
                                    CONFD_ATTR_INACTIVE should be hidden */

    /* ConfD internal fields                              */
    int index;         /* array pos                        */
    int lastop;        /* remember what we were doing      */
    int last_proto_op; /* ditto */
    int seen_reply;    /* have we seen a reply msg         */
    int query_ref;     /* last query ref for this trans    */
    int in_num_instances;
    u_int32_t num_instances;
    long nextarg;
    struct confd_data_cbs *next_dcb;
    confd_hkeypath_t *next_kp;
    struct confd_tr_item *lastack; /* tail of acklist */
    int refc;
};
```

This callback is required to prepare for future read/write operations towards the data source. It could be that a file handle or socket must be established. The place to do that is usually the `init()` callback.

The `init()` callback is conceptually invoked at the start of the transaction, but as an optimization, ConfD will as far as possible delay the actual invocation for a given daemon until it is required. In case of a read-only transaction, or a daemon that is only providing operational data, this can have the result that a daemon will not have any callbacks at all invoked (if none of the data elements that it provides are accessed).

The callback must also indicate to `libconfd` which WORKER_SOCKET should be used for future communications in this transaction. This is the mechanism which is used by libconfd to distribute work among multiple worker threads in the database application. If another thread than the thread which owns the CONTROL_SOCKET should be used, it is up to the application to somehow notify that thread.

The choice of descriptor is done through the API call `confd_trans_set_fd()` which sets the `fd` field in the transaction context.

The callback must return CONFD_OK, CONFD_DELAYED_RESPONSE or CONFD_ERR.

The transaction then enters READ state, where ConfD will perform a series of `read()` operations.

`trans_lock()`

This callback is invoked when the validation phase of the transaction starts. If the underlying database supports real transactions, it is usually appropriate to start such a native transaction here.

The callback must return CONFD_OK, CONFD_DELAYED_RESPONSE, CONFD_ERR, or CONFD_ALREADY_LOCKED. The transaction enters VALIDATE state, where ConfD will perform a series of `read()` operations.

The trans lock is set until either `trans_unlock()` or `finish()` is called. ConfD ensures that a trans_lock is set on a single transaction only. In the case of the CONFD_DELAYED_RESPONSE - to later indicate that the database is already locked, use the `confd_delayed_reply_error()` function with the special error string "locked". An alternate way to indicate that the database is already locked is to use `confd_trans_seterr_extended()` (see below) with CONFD_ERRCODE_IN_USE - this is the only way to give a message in the "delayed" case. If this function is used, the callback must return CONFD_ERR in the "normal" case, and in the "de-

layed" case `confd_delayed_reply_error()` must be called with a NULL argument after `confd_trans_seterr_extended()`.

`trans_unlock()`
This callback is called when the validation of the transaction failed, or the validation is triggered explicitly (i.e. not part of a 'commit' operation). This is common in the CLI and the Web UI where the user can enter invalid data. Transactions that originate from NETCONF will never trigger this callback. If the underlying database supports real transactions and they are used, the transaction should be aborted here.

The callback must return CONFD_OK, CONFD_DELAYED_RESPONSE or CONFD_ERR. The transaction re-enters READ state.

`write_start()`
This callback is invoked when the validation succeeded and the write phase of the transaction starts. If the underlying database supports real transactions, it is usually appropriate to start such a native transaction here.

The transaction enters the WRITE state. No more `read()` operations will be performed by ConfD.

The callback must return CONFD_OK, CONFD_DELAYED_RESPONSE, CONFD_ERR, or CONFD_IN_USE.

If CONFD_IN_USE is returned, the transaction is restarted, i.e. it effectively returns to the READ state. To give this return code after CONFD_DELAYED_RESPONSE, use the `confd_delayed_reply_error()` function with the special error string "in_use". An alternative for both cases is to use `confd_trans_seterr_extended()` (see below) with CONFD_ERRCODE_IN_USE - this is the only way to give a message in the "delayed" case. If this function is used, the callback must return CONFD_ERR in the "normal" case, and in the "delayed" case `confd_delayed_reply_error()` must be called with a NULL argument after `confd_trans_seterr_extended()`.

`prepare()`
If we have multiple sources of data it is highly recommended that the callback is implemented. The callback is called at the end of the transaction, when all read and write operations for the transaction have been performed and the transaction should prepare to commit.

This callback should allocate the resources necessary for the commit, if any. The callback must return CONFD_OK, CONFD_DELAYED_RESPONSE, CONFD_ERR, or CONFD_IN_USE.

If CONFD_IN_USE is returned, the transaction is restarted, i.e. it effectively returns to the READ state. To give this return code after CONFD_DELAYED_RESPONSE, use the `confd_delayed_reply_error()` function with the special error string "in_use". An alternative for both cases is to use `confd_trans_seterr_extended()` (see below) with CONFD_ERRCODE_IN_USE - this is the only way to give a message in the "delayed" case. If this function is used, the callback must return CONFD_ERR in the "normal" case, and in the "delayed" case `confd_delayed_reply_error()` must be called with a NULL argument after `confd_trans_seterr_extended()`.

`commit()`
This callback is optional. This callback is responsible for writing the data to persistent storage. Must return CONFD_OK, CONFD_DELAYED_RESPONSE or CONFD_ERR.

`abort()`
This callback is optional. This callback is responsible for undoing whatever was done in the `prepare()` phase. Must return CONFD_OK, CONFD_DELAYED_RESPONSE or CONFD_ERR.

`finish()`
: This callback is optional. This callback is responsible for releasing resources allocated in the `init()` phase. In particular, if the application choose to use the `t_opaque` field in the confd_trans_ctx to hold any resources, these resources must be released here.

`interrupt()`
: This callback is optional. Unlike the other transaction callbacks, it does not imply a change of the transaction state, it is instead a notification that the user running the transaction requested that it should be interrupted (e.g. Ctrl-C in the CLI). Also unlike the other transaction callbacks, the callback request is sent asynchronously on the control socket. Registering this callback may be useful for a configuration data provider that has some (transaction or data) callbacks which require extensive processing - the callback could then determine whether one of these callbacks is being processed, and if feasible return an error from that callback instead of completing the processing. In that case, `confd_trans_seterr_extended()` with *code* `CONFD_ERRCODE_INTERRUPT` should be used.

All the callback functions (except `interrupt()`) must return CONFD_OK, CONFD_DELAYED_RESPONSE or CONFD_ERR.

It is often useful to associate an error string with a CONFD_ERR return value. This can be done through a call to `confd_trans_seterr()` or `confd_trans_seterr_extended()`.

Depending on the situation (original caller) the error string gets propagated to the CLI, the Web UI or the NETCONF manager.

`int` **`confd_register_db_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_db_cbs *`*`dbcbs`*`);`

We may also optionally have a set of callback functions which span over several ConfD transactions.

If the system is configured in such a way so that the external database owns the candidate data store we must implement four callback functions to do this. If ConfD owns the candidate the candidate callbacks should be set to NULL.

If ConfD owns the candidate, and ConfD has been configured to support `confirmed-commit`, then three checkpointing functions must be implemented; otherwise these should be set to NULL. When `confirmed-commit` is enabled, the user can commit the candidate with a timeout. Unless a confirming commit is given by the user before the timer expires, the system must rollback to the previous running configuration. This mechanism is controlled by the checkpoint callbacks. See further below.

An external database may also (optionally) support the lock/unlock and lock_partial/unlock_partial operations. This is only interesting if there exists additional locking mechanisms towards the database - such as an external CLI which can lock the database, or if the external database owns the candidate.

Finally, the external database may optionally validate a candidate configuration. Configuration validation is preferably done through ConfD - however if a system already has implemented extensive configuration validation - the `candidate_validate()` callback can be used.

The struct confd_db_cbs structure looks like:

```
struct confd_db_cbs {
    int (*candidate_commit)(struct confd_db_ctx *dbx, int timeout);
    int (*candidate_confirming_commit)(struct confd_db_ctx *dbx);
    int (*candidate_reset)(struct confd_db_ctx *dbx);
    int (*candidate_chk_not_modified)(struct confd_db_ctx *dbx);
    int (*candidate_rollback_running)(struct confd_db_ctx *dbx);
    int (*candidate_validate)(struct confd_db_ctx *dbx);
```

```
    int (*add_checkpoint_running)(struct confd_db_ctx *dbx);
    int (*del_checkpoint_running)(struct confd_db_ctx *dbx);
    int (*activate_checkpoint_running)(struct confd_db_ctx *dbx);
    int (*copy_running_to_startup)(struct confd_db_ctx *dbx);
    int (*running_chk_not_modified)(struct confd_db_ctx *dbx);
    int (*lock)(struct confd_db_ctx *dbx, enum confd_dbname dbname);
    int (*unlock)(struct confd_db_ctx *dbx, enum confd_dbname dbname);
    int (*lock_partial)(struct confd_db_ctx *dbx,
                        enum confd_dbname dbname, int lockid,
                        confd_hkeypath_t paths[], int npaths);
    int (*unlock_partial)(struct confd_db_ctx *dbx,
                          enum confd_dbname dbname, int lockid);
    int (*delete_config)(struct confd_db_ctx *dbx,
                         enum confd_dbname dbname);
};
```

If we have an externally implemented candidate, that is if confd.conf item `/confdConfig/datastores/candidate/implementation` is set to "external", we must implement the 5 candidate callbacks. Otherwise (recommended) they must be set to NULL.

If implementation is "external", all databases (if there are more than one) MUST take care of the candidate for their part of the configuration data tree. If ConfD is configured to use an external database for parts of the configuration, and the built-in CDB database is used for some parts, CDB will handle the candidate for its part. See also `misc/extern_candidate` in the examples collection.

The callback functions are are the following:

`candidate_commit()`
: This function should copy the candidate DB into the running DB. If *`timeout`* != 0, we should be prepared to do a rollback or act on a `candidate_confirming_commit()`. The *`timeout`* parameter can not be used to set a timer for when to rollback; this timer is handled by the ConfD daemon. If we terminate without having acted on the `candidate_confirming_commit()`, we MUST restart with a rollback. Thus we must remember that we are waiting for a `candidate_confirming_commit()` and we must do so on persistent storage. Must only be implemented when the external database owns the candidate.

`candidate_confirming_commit()`
: If the *`timeout`* in the `candidate_commit()` function is != 0, we will be either invoked here or in the `candidate_rollback_running()` function within *`timeout`* seconds. `candidate_confirming_commit()` should make the commit persistent, whereas a call to `candidate_rollback_running()` would copy back the previous running configuration to running.

`candidate_rollback_running()`
: If for some reason, apart from a timeout, something goes wrong, we get invoked in the `candidate_rollback_running()` function. The function should copy back the previous running configuration to running.

`candidate_reset()`
: This function is intended to copy the current running configuration into the candidate. It is invoked whenever the NETCONF operation `<discard-changes>` is executed or when a lock is released without committing.

`candidate_chk_not_modified()`
: This function should check to see if the candidate has been modified or not. Returns CONFD_OK if no modifications has been done since the last commit or reset, and CONFD_ERR if any uncommitted modifications exist.

`candidate_validate()`
: This callback is optional. If implemented, the task of the callback is to validate the candidate configuration. Note that the running database can be validated by the database in the `prepare()` callback. `candidate_validate()` is only meaningful when an explicit validate operation is received, e.g. through NETCONF.

`add_checkpoint_running()`
: This function should be implemented only when ConfD owns the candidate, and confirmed-commit is enabled.

  It is responsible for creating a checkpoint of the current running configuration and storing the checkpoint in non-volatile memory. When the system restarts this function should check if there is a checkpoint available, and use the checkpoint instead of running.

`del_checkpoint_running()`
: This function should delete a checkpoint created by `add_checkpoint_running()`. It is called by ConfD when a confirming commit is received.

`activate_checkpoint_running()`
: This function should rollback running to the checkpoint created by `add_checkpoint_running()`. It is called by ConfD when the timer expires or if the user session expires.

`copy_running_to_startup()`
: This function should copy running to startup. It only needs to be implemented if the startup data store is enabled.

`running_chk_not_modified()`
: This function should check to see if running has been modified or not. It only needs to be implemented if the startup data store is enabled. Returns CONFD_OK if no modifications have been done since the last copy of running to startup, and CONFD_ERR if any modifications exist.

`lock()`
: This should only be implemented if our database supports locking from other sources than through ConfD. In this case both the lock/unlock and lock_partial/unlock_partial callbacks must be implemented. If a lock on the whole database is set through e.g. NETCONF, ConfD will first make sure that no other ConfD transaction has locked the database. Then it will call `lock()` to make sure that the database is not locked by some other source (such as a non-ConfD CLI). Returns CONFD_OK on success, and CONFD_ERR if the lock was already held by an external entity.

`unlock()`
: Unlocks the database.

`lock_partial()`
: This should only be implemented if our database supports locking from other sources than through ConfD, see `lock()` above. This callback is invoked if a northbound agent requests a partial lock. The *`paths[]`* argument is an *`npaths`* long array of hkeypaths that identify the leafs and/or subtrees that are to be locked. The *`lockid`* is a reference that will be used on a subsequent corresponding `unlock_partial()` invocation.

`unlock_partial()`
: Unlocks the partial lock that was requested with *`lockid`*.

`delete_config()`
: Will be called for 'startup' or 'candidate' only. The database is supposed to be set to erased.

All the above callback functions must return either CONFD_OK or CONFD_ERR. If the system is configured so that ConfD owns the candidate, then obviously the candidate related functions need not be implemented. If the system is configured to not do confirmed commit, `candidate_confirming_commit()` and `candidate_commit()` need not to be implemented.

It is often interesting to associate an error string with a CONFD_ERR return value. In particular the `validate()` callback must typically indicate which item was invalid and why. This can be done through a call to `confd_db_seterr()` or `confd_db_seterr_extended()`.

Depending on the situation (original caller) the error string is propagated to the CLI, the Web UI or the NETCONF manager.

`int` **`confd_register_data_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_data_cbs *`*`data`*`);`

This function registers the data manipulation callbacks. The data model defines a number of "callpoints". Each callpoint must have an associated set of data callbacks.

Thus if our database application serves three different callpoints in the data model we must install three different sets of data manipulation callbacks - one set at each callpoint.

The data callbacks either return data back to ConfD or they do not. For example the `create()` callback does not return data whereas the `get_next()` callback does. All the callbacks that return data do so through API functions, not by means of return values from the function itself.

The struct confd_data_cbs is defined as:

```
struct confd_data_cbs {
    char callpoint[MAX_CALLPOINT_LEN];
    /* where in the XML tree do we */
    /* want this struct */

    /* Only necessary to have this cb if our data model has */
    /* typeless optional nodes or oper data lists w/o keys */
    int (*exists_optional)(struct confd_trans_ctx *tctx,
                           confd_hkeypath_t *kp);
    int (*get_elem)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp);
    int (*get_next)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp, long next);
    int (*set_elem)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp,
                    confd_value_t *newval);
    int (*create)(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *kp);
    int (*remove)(struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *kp);
    /* optional (find list entry by key/index values) */
    int (*find_next)(struct confd_trans_ctx *tctx,
                     confd_hkeypath_t *kp,
                     enum confd_find_next_type type,
                     confd_value_t *keys, int nkeys);
    /* optional optimizations */
    int (*num_instances)(struct confd_trans_ctx *tctx,
                         confd_hkeypath_t *kp);
    int (*get_object)(struct confd_trans_ctx *tctx,
                      confd_hkeypath_t *kp);
    int (*get_next_object)(struct confd_trans_ctx *tctx,
                           confd_hkeypath_t *kp, long next);
```

```
    int (*find_next_object)(struct confd_trans_ctx *tctx,
                            confd_hkeypath_t *kp,
                            enum confd_find_next_type type,
                            confd_value_t *keys, int nkeys);
    /* next two are only necessary if 'choice' is used */
    int (*get_case)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp, confd_value_t *choice);
    int (*set_case)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp, confd_value_t *choice,
                    confd_value_t *caseval);
    /* next two are only necessary for config data providers,
       and only if /confdConfig/enableAttributes is 'true' */
    int (*get_attrs)(struct confd_trans_ctx *tctx,
                     confd_hkeypath_t *kp,
                     u_int32_t *attrs, int num_attrs);
    int (*set_attr)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp,
                    u_int32_t attr, confd_value_t *v);
    /* only necessary if "ordered-by user" is used */
    int (*move_after)(struct confd_trans_ctx *tctx,
                      confd_hkeypath_t *kp, confd_value_t *prevkeys);
    /* only for per-transaction-invoked transaction hook */
    int (*write_all)(struct confd_trans_ctx *tctx,
                     confd_hkeypath_t *kp);
    void *cb_opaque; /* private user data    */
};
```

One of the parameters to the callback is a confd_hkeypath_t (h - as in hashed keypath). This is fully described in confd_types(3).

The `cb_opaque` element can be used to pass arbitrary data to the callbacks, e.g. when the same set of callbacks is used for multiple callpoints. It is made available to the callbacks via an element with the same name in the transaction context (*`tctx`* argument), see the structure definition above.

If the `tailf:opaque` substatement has been used with the `tailf:callpoint` statement in the data model, the argument string is made available to the callbacks via the `callpoint_opaque` element in the transaction context.

When use of the `CONFD_ATTR_INACTIVE` attribute is enabled in the ConfD configuration (`/confdConfig/enableAttributes` and `/confdConfig/enableInactive` both set to `true`), read callbacks (`get_elem()` etc) for configuration data must observe the current value of the `hide_inactive` element in the transaction context. If it is non-zero, those callbacks must act as if data with the `CONFD_ATTR_INACTIVE` attribute set does not exist.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE

`get_elem()`

This callback function needs to return the value of a specific leaf. Assuming we have the following data model:

```
container servers {
  tailf:callpoint mycp;
  list server {
    key name;
    max-elements 64;
    leaf name {
      type string;
    }
    leaf ip {
```

```
      type inet:ip-address;
    }
    leaf port {
      type inet:port-number;
    }
  }
}
```

For example the value of the `ip` leaf in the server entry whose key is "www" can be returned separately. The way to return a single data item is through `confd_data_reply_value()`.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available. In the latter case the application must at a later stage call `confd_data_reply_value()` (or `confd_delayed_reply_ok()` for a write operation). If an error is discovered at the time of a delayed reply, the error is signaled through a call to `confd_delayed_reply_error()`

If the leaf does not exist the callback must call `confd_data_reply_not_found()`. If the leaf has a default value defined in the data model, and no value has been set, the callback should use `confd_data_reply_value()` with a value of type C_DEFAULT - this makes it possible for northbound agents to leave such leafs out of the data returned to the user/manager (if requested).

The implementation of `get_elem()` must be prepared to return values for all the leafs including the key(s). When ConfD invokes `get_elem()` on a key leaf it is an existence test. The application should verify whether the object exists or not.

`get_next()`

This callback makes it possible for ConfD to traverse a set of list entries. The *`next`* parameter will be `-1` on the first invocation. This function should reply by means of the function `confd_data_reply_next_key()`.

If the list has a `tailf:secondary-index` statement (see tailf_yang_extensions(5)), and the entries are supposed to be retrieved according to one of the secondary indexes, the variable `tctx->secondary_index` will be set to a value greater than `0`, indicating which secondary-index is used. The first secondary-index in the definition is identified with the value `1`, the second with `2`, and so on. confdc can be used to generate `#define`s for the index names. If no secondary indexes are defined, or if the sort order should be according to the key values, `tctx->secondary_index` is `0`.

To signal that no more entries exist, we reply with a NULL pointer as the key value in the `confd_data_reply_next_key()` function.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available. In the latter case the application must at a later stage call `confd_data_reply_next_key()`.

Note

For a list that does not specify a non-default sort order by means of a `ordered-by user` or `tailf:sort-order` statement, ConfD assumes that list entries are ordered strictly by increasing key (or secondary index) values. Thus for correct operation, we must observe this order when returning list entries in a sequence of `get_next()` calls.

`set_elem()`

This callback writes the value of a leaf. Note that an optional leaf with a type other than empty is created by a call to this function. The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE.

`create()`
: This callback creates a new list entry, a `presence` container, or a leaf of type `empty`. In the case of the `servers` data model above, this function need to create a new `server` entry. Must return CONFD_OK on success, CONFD_ERR on error, CONFD_DELAYED_RESPONSE or CONFD_ACCUMULATE.

  The data provider is responsible for maintaining the order of list entries. If the list is marked as `ordered-by user` in the YANG data model, the `create()` callback must add the list entry to the end of the list.

`remove()`
: This callback is used to remove an existing list entry, a `presence` container, or an optional leaf and all its sub nodes (if any). When we use the YANG `choice` statement in the data model, it may also be used to remove nodes that are not optional as such when a different `case` (or none) is selected. I.e. it must always be possible to remove cases in a choice.

  Must return CONFD_OK on success, CONFD_ERR on error, CONFD_DELAYED_RESPONSE or CONFD_ACCUMULATE.

`exists_optional()`
: If we have `presence` containers or leafs of type `empty`, we cannot use the `get_elem()` callback to read the value of such a node, since it does not have a type. An example of a data model could be:

```
container bs {
  presence "";
  tailf:callpoint bcp;
  list b {
    key name;
    max-elements 64;
    leaf name {
      type string;
    }
    container opt {
      presence "";
      leaf ii {
        type int32;
      }
    }
    leaf foo {
      type empty;
    }
  }
}
```

  The above YANG fragment has 3 nodes that may or may not exist and that do not have a type. If we do not have any such elements, nor any operational data lists without keys (see below), we do not need to implement the `exists_optional()` callback and can set it to NULL.

  If we have the above data model, we must implement the `exists_optional()`, and our implementation must be prepared to reply on calls of the function for the paths `/bs`, `/bs/b/opt`, and `/bs/b/foo`. The leaf `/bs/b/opt/ii` is not mandatory, but it does have a type namely int32, and thus the existence of that leaf will be determined through a call to the `get_elem()` callback.

  The `exists_optional()` callback may also be invoked by ConfD as "existence test" for an entry in an operational data list without keys (see the Operational Data chapter in the User Guide). Normally this existence test is done with a `get_elem()` request for the first key, but since there are no keys, this callback is used instead. Thus if we have such lists, we must also implement this callback, and handle a request where the keypath identifies a list entry.

The callback must reply to ConfD using either the `confd_data_reply_not_found()` or the `confd_data_reply_found()` function.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available.

`find_next()`

This optional callback can be registered to optimize cases where ConfD wants to start a list traversal at some other point than at the first entry of the list, or otherwise make a "jump" in a list traversal. If the callback is not registered, ConfD will use a sequence of `get_next()` calls to find the desired list entry.

Where the `get_next()` callback provides a *`next`* parameter to indicate which keys should be returned, this callback instead provides a *`type`* parameter and a set of values to indicate which keys should be returned. Just like for `get_next()`, the callback should reply by calling `confd_data_reply_next_key()` with the keys for the requested list entry.

The *`keys`* parameter is a pointer to a *`nkeys`* elements long array of key values, or secondary index-leaf values (see below). The *`type`* can have one of two values:

`CONFD_FIND_NEXT` — The callback should always reply with the key values for the first list entry *after* the one indicated by the *`keys`* array, and a *`next`* value appropriate for retrieval of subsequent entries. The *`keys`* array may not correspond to an actual existing list entry - the callback must return the keys for the first existing entry that is "later" in the list order than the keys provided by the callback. Furthermore the number of values provided in the array (*`nkeys`*) may be fewer than the number of keys (or number of index-leafs for a secondary-index) in the data model, possibly even zero. This means that only the first *`nkeys`* values are provided, and the remaining ones should be taken to have a value "earlier" than the value for any existing list entry.

`CONFD_FIND_SAME_OR_NEXT` — If the values in the *`keys`* array completely identify an actual existing list entry, the callback should reply with the keys for this list entry and a corresponding *`next`* value. Otherwise the same logic as described for `CONFD_FIND_NEXT` should be used.

The `dp/find_next` example in the bundled examples collection has an implementation of the `find_next()` callback for a list with two integer keys. It shows how the *`type`* value and the provided keys need to be combined in order to find the requested entry - or find that no entry matching the request exists.

If the list has a `tailf:secondary-index` statement (see tailf_yang_extensions(5)), the callback must examine the value of the `tctx->secondary_index` variable, as described for the `get_next()` callback. If `tctx->secondary_index` has a value greater than `0`, the *`keys`* and *`nkeys`* parameters do not represent key values, but instead values for the index leafs specified by the `tailf:index-leafs` statement for the secondary index. The callback should however still reply with the actual key values for the list entry in the `confd_data_reply_next_key()` call.

Once we have called `confd_data_reply_next_key()`, ConfD will use `get_next()` (or `get_next_object()`) for any subsequent entry-by-entry list traversal - however we can request that this traversal should be done using `find_next()` (or `find_next_object()`) instead, by passing `-1` for the *`next`* parameter to `confd_data_reply_next_key()`. In this case ConfD will always invoke `find_next()`/`find_next_object()` with *`type`* `CONFD_FIND_NEXT`, and the (complete) set of keys from the previous reply.

Note

In the case of list traversal by means of a secondary index, the secondary index values must be unique for entry-by-entry traversal with `find_next()`/ `find_next_object()` to be possible. Thus we can not pass `-1` for the *`next`* parameter to `confd_data_reply_next_key()` in this case if the secondary index values are not unique.

To signal that no entry matching the request exists, i.e. we have reached the end of the list while evaluating the request, we reply with a NULL pointer as the key value in the `confd_data_reply_next_key()` function.

Note

For a list that does not specify a non-default sort order by means of a `ordered-by user` or `tailf:sort-order` statement, ConfD assumes that list entries are ordered strictly by increasing key values.

If we have registered `find_next()` (or `find_next_object()`), it is not strictly necessary to also register `get_next()` (or `get_next_object()`) - except for the case of traversal by secondary index when the secondary index values are not unique, see above. If a northbound agent does a get_next request, and neither `get_next()` nor `get_next_object()` is registered, ConfD will instead invoke `find_next()` (or `find_next_object()`), the same way as if `-1` had been passed for the *`next`* parameter to `confd_data_reply_next_key()` as described above - the actual *`next`* value passed is ignored. The very first get_next request for a traversal (i.e. where the *`next`* parameter would be `-1`) will cause a find_next invocation with *`type`* `CONFD_FIND_NEXT` and *`nkeys`* == 0, i.e. no keys provided.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available. In the latter case the application must at a later stage call `confd_data_reply_next_key()`.

`num_instances()`

This callback can optionally be implemented. The purpose is to return the number of entries in a list. If the callback is set to NULL, whenever ConfD needs to calculate the number of entries in a certain list, ConfD will iterate through the entries by means of consecutive calls to the `get_next()` callback.

If we have a large number of entries *and* it is computationally cheap to calculate the number of entries in a list, it may be worth the effort to implement this callback for performance reasons.

The number of entries is returned in an confd_value_t value of type C_INT32. The value is returned through a call to `confd_data_reply_value()`, see code example below:

```
    int num_instances;
    confd_value_t v;

    CONFD_SET_INT32(&v, num_instances);
    confd_data_reply_value(trans_ctx, &v);
    return CONFD_OK;
```

Must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE.

`get_object()`

The implementation of this callback is also optional. The purpose of the callback is to return an entire object, i.e. a list entry, in one swoop. If the callback is not implemented, ConfD will retrieve the whole object through a series of calls to `get_elem()`.

The callback will only be called for list entries - i.e. `get_elem()` is still needed for leafs that are not defined in a list, but if there are no such leafs in the part of the data model covered by a given callpoint, the `get_elem()` callback may be omitted when `get_object()` is registered. This has the drawback that ConfD will have to invoke get_object() even if only a single leaf in a list entry is needed though, e.g. for the existence test mentioned for `get_elem()`.

When ConfD invokes the `get_elem()` callback, it is the responsibility of the application to issue calls to the reply function `confd_data_reply_value()`. The `get_object()` callback cannot use this function since it needs to return a sequence of values. The `get_object()` callback must use either the `confd_data_reply_value_array()` function or the `confd_data_reply_tag_value_array()` function. See the description of these functions below for the details of the arguments passed. If the entry requested does not exist, the callback must call `confd_data_reply_not_found()`.

Remember, the callback `exists_optional()` must always be implemented when we have `presence` containers or leafs of type `empty`. If we also choose to implement the `get_object()` callback, ConfD can sometimes derive the existence of such a node through a previous call to `get_object()`. This is however not always the case, thus even if we implement `get_object()`, we must also implement `exists_optional()`if we have such nodes.

If we pass an array of values which does not comply with the rules for the above functions, ConfD will notice and an error is reported to the agent which issued the request. A message is also logged to ConfD's developerLog.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available.

`get_next_object()`

The implementation of this callback is also optional. Similar to the `get_object()` callback the purpose of this callback is to return an entire object, or even multiple objects, in one swoop. It combines the functionality of `get_next()` and `get_object()` into a single callback, and adds the possibility to return multiple objects. Thus we need only implement this callback if it very important to be able to traverse a list very fast. If the callback is not implemented, ConfD will retrieve the whole object through a series of calls to `get_next()` and consecutive calls to either `get_elem()` or `get_object()`.

When we have registered `get_next_object()`, it is not strictly necessary to also register `get_next()`, but omitting `get_next()` may have a serious performance impact, since there are cases (e.g. CLI tab completion) when ConfD only wants to retrieve the keys for a list. In such a case, if we have only registered `get_next_object()`, all the data for the list will be retrieved, but everything except the keys will be discarded. Also note that even if we have registered `get_next_object()`, at least one of the `get_elem()` and `get_object()` callbacks must be registered.

Similar to the `get_next()` callback, if the *`next`* parameter is `-1` ConfD wants to retrieve the first entry in the list.

Similar to the `get_next()` callback, if the `tctx->secondary_index` parameter is greater than `0` ConfD wants to retrieve the entries in the order defined by the secondary index.

Similar to the `get_object()` callback, `get_next_object()` needs to reply with an entire object expressed as either an array of confd_value_t values or an array of confd_tag_value_t values. It must also indicate which is the *next* entry in the list similar to the `get_next()` callback. The two functions `confd_data_reply_next_object_array()` and `confd_data_reply_next_object_tag_value_array()` are use to convey the return values for one object from the `get_next_object()` callback.

If we want to reply with multiple objects, we must instead use one of the functions `confd_data_reply_next_object_arrays()` and `confd_data_reply_next_object_tag_value_arrays()`. These functions take an "array of object arrays", where each element in the array corresponds to the reply for a single object with `confd_data_reply_next_object_array()` and `confd_data_reply_next_object_tag_value_array()`, respectively.

If we pass an array of values which does not comply with the rules for the above functions, ConfD will notice and an error is reported to the agent which issued the request. A message is also logged to ConfD's developerLog.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available.

`find_next_object()`

The implementation of this callback is also optional. It relates to `get_next_object()` in exactly the same way as `find_next()` relates to `get_next()`. I.e. instead of a parameter *`next`*, we get a *`type`* parameter and a set of key values, or secondary index-leaf values, to indicate which object or objects to return to ConfD via one of the reply functions.

Similar to the `get_next_object()` callback, if the `tctx->secondary_index` parameter is greater than `0` ConfD wants to retrieve the entries in the order defined by the secondary index. And as described for the `find_next()` callback, in this case the *`keys`* and *`nkeys`* parameters represent values for the index leafs specified by the `tailf:index-leafs` statement for the secondary index.

Similar to the `get_next_object()` callback, the callback can use any of the functions `confd_data_reply_next_object_array()`, `confd_data_reply_next_object_tag_value_array()`, `confd_data_reply_next_object_arrays()`, and `confd_data_reply_next_object_tag_value_arrays()` to return one or more objects to ConfD.

If we pass an array of values which does not comply with the rules for the above functions, ConfD will notice and an error is reported to the agent which issued the request. A message is also logged to ConfD's developerLog.

The callback must return CONFD_OK on success, CONFD_ERR on error or CONFD_DELAYED_RESPONSE if the reply value is not yet available.

`get_case()`

This callback only needs to be implemented if we use the YANG `choice` statement in the part of the data model that our data provider is responsible for, but when we use choice, the callback is required. It should return the currently selected `case` for the choice given by the *`choice`* argument - *`kp`* is the path to the container or list entry where the choice is defined.

In the general case, where there may be multiple levels of `choice` statements without intervening `container` or `list` statements in the data model, the choice is represented as an array of confd_value_t elements with the type C_XMLTAG, terminated by an element with the type C_NOEXISTS. This array gives a reversed path with alternating choice and case names, from the data node given by *`kp`* to the specific choice that the callback request pertains to - similar to how a confd_hkeypath_t gives a path through the data tree.

If we don't have such "nested" choices in the data model, we can ignore this array aspect, and just treat the *`choice`* argument as a single confd_value_t value. The case is always represented as a confd_value_t with the type C_XMLTAG. I.e. we can use CONFD_GET_XMLTAG() to get the choice tag from *`choice`* and CONFD_SET_XMLTAG() to set the case tag for the reply value.

The callback should use `confd_data_reply_value()` to return the case value to ConfD, or `confd_data_reply_not_found()` for an optional choice without default case if no case is currently selected. If an optional choice with default case does not have a selected case, the callback should use `confd_data_reply_value()` with a value of type C_DEFAULT.

Must return CONFD_OK on success, CONFD_ERR on error, or CONFD_DELAYED_RESPONSE.

`set_case()`

This callback is completely optional, and will only be invoked (if registered) if we use the YANG `choice` statement and provide configuration data. The callback sets the currently selected `case` for the choice given by the *`kp`* and *`choice`* arguments, and is mainly intended to make it easier to support the `get_case()` callback. ConfD will additionally invoke the `remove()` callback for all nodes in the previously selected case, i.e. if we register `set_case()`, we do not need to analyze `set_elem()` callbacks to determine the currently selected case, or figure out which nodes that should be deleted.

For a choice without a `mandatory true` statement, it is possible to have no case at all selected. To indicate that the previously selected case should be deleted without selecting another case, the callback will be invoked with NULL for the *`caseval`* argument.

The callback must return CONFD_OK on success, CONFD_ERR on error, CONFD_DELAYED_RESPONSE or CONFD_ACCUMULATE.

`get_attrs()`

This callback only needs to be implemented for callpoints specified for configuration data, and only if attributes are enabled in the ConfD configuration (`/confdConfig/enableAttributes` set to `true`). These are the currently supported attributes:

```
/* CONFD_ATTR_TAGS: value is C_LIST of C_BUF/C_STR */
#define CONFD_ATTR_TAGS       0x80000000
/* CONFD_ATTR_ANNOTATION: value is C_BUF/C_STR */
#define CONFD_ATTR_ANNOTATION 0x80000001
/* CONFD_ATTR_INACTIVE: value is C_BOOL 1 (i.e. "true") */
#define CONFD_ATTR_INACTIVE   0x00000000
```

The *`attrs`* parameter is an array of attributes of length *`num_attrs`*, giving the requested attributes - if *`num_attrs`* is 0, all attributes are requested. If the node given by *`kp`* does not exist, the callback should reply by calling `confd_data_reply_not_found()`, otherwise it should call `confd_data_reply_attrs()`, even if no attributes are set.

Note

It is very important to observe this distinction, i.e. to use `confd_data_reply_not_found()` when the node doesn't exist, since ConfD may use `get_attrs()` as an existence check when attributes are enabled. (This avoids doing one callback request for existence check and another to collect the attributes.)

Must return CONFD_OK on success, CONFD_ERR on error, or CONFD_DELAYED_RESPONSE.

`set_attr()`

This callback also only needs to be implemented for callpoints specified for configuration data, and only if attributes are enabled in the ConfD configuration (`/confdConfig/enableAttributes` set to `true`). See `get_attrs()` above for the supported attributes.

The callback should set the attribute *attr* for the node given by *kp* to the value *v*. If the callback is invoked with NULL for the value argument, it means that the attribute should be deleted.

The callback must return CONFD_OK on success, CONFD_ERR on error, CONFD_DELAYED_RESPONSE or CONFD_ACCUMULATE.

`move_after()`
This callback only needs to be implemented if we provide configuration data that has YANG lists with a `ordered-by user` statement. The callback moves the list entry given by *kp*. If *prevkeys* is NULL, the entry is moved first in the list, otherwise it is moved after the entry given by *prevkeys*. In this case *prevkeys* is a pointer to an array of key values identifying an entry in the list. The array is terminated with an element that has type C_NOEXISTS.

The callback must return CONFD_OK on success, CONFD_ERR on error, CONFD_DELAYED_RESPONSE or CONFD_ACCUMULATE.

`write_all()`
This callback will only be invoked for a transaction hook specified with `tailf:invocation-mode per-transaction;` - see the chapter Transformations, Hooks, Hidden Data and Symlinks in the User Guide. It is also the only callback that is invoked for such a hook. The callback is expected to make all the modifications to the current transaction that hook functionality requires. The *kp* parameter is currently always NULL, since the callback does not pertain to any particular data node.

The callback must return CONFD_OK on success, CONFD_ERR on error, or CONFD_DELAYED_RESPONSE.

The six write callbacks (excluding `write_all()`), namely `set_elem()`, `create()`, `remove()`, `set_case()`, `set_attr()`, and `move_after()` may return the value CONFD_ACCUMULATE. If CONFD_ACCUMULATE is returned the library will accumulate the written values as a linked list of operations. This list can later be traversed in either of the transaction callbacks `prepare()` or `commit()`.

This provides trivial transaction support for applications that want to implement the ConfD two-phase commit protocol but lacks an underlying database with proper transaction support. The write operations are available as a linked list of confd_tr_item structs:

```
struct confd_tr_item {
    char *callpoint;
    enum confd_tr_op op;
    confd_hkeypath_t *hkp;
    confd_value_t *val;
    confd_value_t *choice;  /* only for set_case */
    u_int32_t attr;         /* only for set_attr */
    struct confd_tr_item *next;
};
```

The list is available in the transaction context in the field `accumulated`. The entire list and its content will be automatically freed by the library once the transaction finishes.

`int` **`confd_register_range_data_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_data_cbs *`*`data`*`, const confd_value_t *`*`lower`*`, const confd_value_t *`*`upper`*`, int `*`numkeys`*`, const char *`*`fmt`*`, ...);`

This is a variant of `confd_register_data_cb()` which registers a set of callbacks for a range of list entries. There can thus be multiple sets of C functions registered on the same callpoint, even by different daemons. The *lower* and *upper* parameters are two *numkeys* long arrays of key values, which define the endpoints of the list range. It is also possible to do a "default" registration, by giving *lower* and

upper as NULL (*numkeys* is ignored). The callbacks for the default registration will be invoked when the keys are not in any of the explicitly registered ranges.

The *fmt* and remaining parameters specify a string path for the list that the keys apply to, in the same form as for the confd_lib_maapi(3) and confd_lib_cdb(3) functions. However if the list is a sublist to another list, the key element for the parent list(s) may be completely omitted, to indicate that the registration applies to all entries for the parent list(s) (similar to CDB subscription paths).

An example that registers one set of callbacks for the range `/servers/server{aaa}` - `/servers/server{mzz}` and another set for `/servers/server{naa}` - `/servers/server{zzz}`:

```
confd_value_t lower, upper;

CONFD_SET_STR(&lower, "aaa");
CONFD_SET_STR(&upper, "mzz");
if (confd_register_range_data_cb(dctx, &data_cb1, &lower, &upper, 1,
                                 "/servers/server") == CONFD_ERR)
    confd_fatal("Failed to register data cb\n");

CONFD_SET_STR(&lower, "naa");
CONFD_SET_STR(&upper, "zzz");
if (confd_register_range_data_cb(dctx, &data_cb2, &lower, &upper, 1,
                                 "/servers/server") == CONFD_ERR)
    confd_fatal("Failed to register data cb\n");
```

In this example, as in most cases where this function is used, the data model defines a list with a single key, and *numkeys* is thus always `1`. However it can also be used for lists that have multiple keys, in which case the *upper* and *lower* arrays may be populated with multiple keys, upto however many keys the data model specifies for the list, and *numkeys* gives the number of keys in the arrays. If fewer keys than specified in the data model are given, the registration covers all possible values for the remaining keys, i.e. they are effectively wildcarded.

While traversal of a list with range registrations will always invoke e.g. `get_next()` only for actually registered ranges, it is also possible that a request from a northbound interface is made for data in a specific list entry. If the registrations do not cover all possible key values, such a request could be for a list entry that does not fall in any of the registered ranges, which will result in a "no registration" error. To avoid the error, we can either restrict the type of the keys such that only values that fall in the registered ranges are valid, or, for operational data, use a "default" registration as described above. In this case the daemon with the "default" registration would just reply with `confd_data_reply_not_found()` for all requests for specific data, and `confd_data_reply_next_key()` with NULL for the key values for all `get_next()` etc requests.

Note

For a given callpoint name, there can only be either one non-range registration or a number of range registrations that all pertain to the same list. If a range registration is done after a non-range registration or vice versa, or if a range registration is done with a different list path than earlier range registrations, the latest registration completely replaces the earlier one(s). If we want to register for the same ranges in different lists, we must thus have a unique callpoint for each list.

Note

Range registrations can not be used for lists that have the `tailf:secondary-index` extension, since there is no way for ConfD to traverse the registrations in secondary-index order.

`int` **`confd_register_usess_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_usess_cbs *`*`ucb`*`);`

This function can be used to register information callbacks that are invoked for user session start and stop. The struct confd_usess_cbs is defined as:

```
struct confd_usess_cbs {
    void (*start)(struct confd_daemon_ctx *dx,
                  struct confd_user_info *uinfo);
    void (*stop)(struct confd_daemon_ctx *dx,
                 struct confd_user_info *uinfo);
};
```

Both callbacks are optional. They can be used e.g. for a multi-threaded daemon to manage a pool of worker threads, by allocating worker threads to user sessions. In this case we would ideally allocate a worker thread the first time an `init()` callback for a given user session requires a worker socket to be assigned, and use only the `stop()` usess callback to release the worker thread - using the `start()` callback to allocate a worker thread would often mean that we allocated a thread that was never used. The `u_opaque` element in the struct confd_user_info can be used to manage such allocations.

Note

These callbacks will only be invoked if the daemon has also registered other callbacks. Furthermore, as an optimization, ConfD will delay the invocation of the `start()` callback until some other callback is invoked. This means that if no other callbacks for the daemon are invoked for the duration of a user session, neither `start()` nor `stop()` will be invoked for that user session. If we want timely notification of start and stop for all user sessions, we can subscribe to `CONFD_NOTIF_AUDIT` events, see confd_lib_events(3).

Note

When we call `confd_register_done()` (see below), the `start()` callback (if registered) will be invoked for each user session that already exists.

`int` **`confd_register_done`**`(struct confd_daemon_ctx *`*`dx`*`);`

When we have registered all the callbacks for a daemon (including the other types described below if we have them), we must call this function to synchronize with ConfD. No callbacks will be invoked until it has been called, and after the call, no further registrations are allowed.

`int` **`confd_fd_ready`**`(struct confd_daemon_ctx *`*`dx`*`, int `*`fd`*`);`

The database application owns all data provider sockets to ConfD and is responsible for the polling of these sockets. When one of the ConfD sockets has I/O ready to read, the application must invoke `confd_fd_ready()` on the socket. This function will:

- Read data from ConfD
- Unmarshal this data
- Invoke the right callback with the right arguments

When this function reads the request from from ConfD it will block on `read()`, thus if it is important for the application to have nonblocking I/O, the application must dispatch I/O from ConfD in a separate thread.

The function returns the return value from the callback function, normally CONFD_OK (0), or CONFD_ERR (-1) on error and CONFD_EOF (-2) when the socket to ConfD has been closed. Thus CONFD_ERR can mean either that the callback function that was invoked returned CONFD_ERR, or that some error condition occurred within the `confd_fd_ready()` function. These cases can be dis-

tinguished via `confd_errno`, which will be set to CONFD_ERR_EXTERNAL if CONFD_ERR comes from the callback function. Thus a correct call to `confd_fd_ready()` looks like:

```
struct pollfd set[n];
/* ..... */

if (set[0].revents & POLLIN) {
    if ((ret = confd_fd_ready(dctx, mysock)) == CONFD_EOF) {
        confd_fatal("ConfD socket closed\n");
    } else if (ret == CONFD_ERR &&
               confd_errno != CONFD_ERR_EXTERNAL) {
        confd_fatal("Error on ConfD socket request: %s (%d): %s\n",
                    confd_strerror(confd_errno), confd_errno,
                    confd_lasterr());
    }
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE, CONFD_ERR_EXTERNAL

`void` **`confd_trans_set_fd`**`(struct confd_trans_ctx *`*`tctx`*`, int `*`sock`*`);`

Associate a worker socket with the transaction, or validation phase. This function must be called in the transaction and validation `init()` callbacks - a minimal implementation of a transaction `init()` callback looks like:

```
static int init(struct confd_trans_ctx *tctx)
{
    confd_trans_set_fd(tctx, workersock);
    return CONFD_OK;
}
```

`int` **`confd_data_reply_value`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_value_t *`*`v`*`);`

This function is used to return a single data item to ConfD.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_value_array`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_value_t *`*`vs`*`, int `*`n`*`);`

This function is used to return an array of values, corresponding to a complete list entry, to ConfD. It can be used by the optional `get_object()` callback. The *`vs`* array is populated with *`n`* values according to the specification of the Value Array format in the XML STRUCTURES section of the confd_types(3) manual page.

In the easiest case, similar to the "servers" example above, we can construct a reply array as follows:

```
struct in_addr ip4 = my_get_ip(.....);
confd_value_t ret[3];

CONFD_SET_STR(&ret[0], "www");
CONFD_SET_IPV4(&ret[1], ip4);
CONFD_SET_UINT16(&ret[2], 80);
confd_data_reply_value_array(tctx, ret, 3);
```

Any containers inside the object must also be passed in the array. For example an entry in the `b` list used in the explanation for `exists_optional()` would have to be passed as:

```
confd_value_t ret[4];

CONFD_SET_STR(&ret[0], "b_name");
CONFD_SET_XMLTAG(&ret[1], myprefix_opt, myprefix__ns);
CONFD_SET_INT32(&ret[2], 77);
CONFD_SET_NOEXISTS(&ret[3]);

confd_data_reply_value_array(tctx, ret, 4);
```

Thus, a container or a leaf of type `empty` must be passed as its equivalent XML tag if it exists. If a `presence` container or leaf of type `empty` does not exist, it must be passed as a value of C_NOEXISTS. In the example above, the leaf `foo` does not exist, thus the contents of position `3` in the array.

If a `presence` container does not exist, its non existing values must not be passed - it suffices to say that the container itself does not exist. In the example above, the `opt` container did exist and thus we also had to pass the contained value(s), the `ii` leaf.

Hence, the above example represents:

```
<b>
   <name>b_name</name>
   <opt>
      <ii>77</ii>
   </opt>
</b>
```

`int` **`confd_data_reply_tag_value_array`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_tag_value_t *`*`tvs`*`, int `*`n`*`);`

This function is used to return an array of values, corresponding to a complete list entry, to ConfD. It can be used by the optional `get_object()` callback. The *`tvs`* array is populated with *`n`* values according to the specification of the Tagged Value Array format in the XML STRUCTURES section of the confd_types(3) manual page.

I.e. the difference from `confd_data_reply_value_array()` is that the values are tagged with the node names from the data model - this means that non-existing values can simply be omitted from the array, per the specification above. Additionally the key leafs can be omitted, since they are already known by ConfD - if the key leafs are included, they will be ignored. Finally, in e.g. the case of a container with both config and non-config data, where the config data is in CDB and only the non-config data provided by the callback, the config elements can be omitted (for `confd_data_reply_value_array()` they must be included as C_NOEXISTS elements).

However, although the tagged value array format can represent nested lists, these must not be passed via this function, since the `get_object()` callback only pertains to a single entry of one list. Nodes representing sub-lists must thus be omitted from the array, and ConfD will issue separate `get_object()` invocations to retrieve the data for those.

Using the same examples as above, in the "servers" case, we can construct a reply array as follows:

```
struct in_addr ip4 = my_get_ip(.....);
confd_tag_value_t ret[2];
int n = 0;

CONFD_SET_TAG_IPV4(&ret[n], myprefix_ip, ip4); n++;
CONFD_SET_TAG_UINT16(&ret[n], myprefix_port, 80); n++;
confd_data_reply_tag_value_array(tctx, ret, n);
```

An entry in the `b` list used in the explanation for `exists_optional()` would be passed as:

```
confd_tag_value_t ret[3];
int n = 0;

CONFD_SET_TAG_XMLBEGIN(&ret[n], myprefix_opt, myprefix__ns); n++;
CONFD_SET_TAG_INT32(&ret[n], myprefix_ii, 77); n++;
CONFD_SET_TAG_XMLEND(&ret[n], myprefix_opt, myprefix__ns); n++;
confd_data_reply_tag_value_array(tctx, ret, n);
```

The C_XMLEND element is not strictly necessary in this case, since there are no subsequent elements in the array. However it would have been required if the optional `foo` leaf had existed, thus it is good practice to always include both the C_XMLBEGIN and C_XMLEND elements for nested containers (if they exist, that is - otherwise neither must be included).

`int` **`confd_data_reply_next_key`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_value_t *`*`v`*`, int `*`num_vals_in_key`*`, long `*`next`*`);`

This function is used by the `get_next()` and `find_next()` callbacks to return the next key. A list may have multiple key leafs specified in the data model. The parameter *`num_vals_in_key`* indicates the number of key values, i.e. the length of the *`v`* array. In the typical case with all lists having just a single key leaf specified, *`num_vals_in_key`* is always 1.

The *`long next`* will be passed into the next invocation of the `get_next()` callback if it has a value other than `-1`. Thus this value provides a means for the application to traverse the data. Since this is long it is possible to pass a void* pointing to the next list entry in the application - effectively passing a pointer to confd and getting it back in the next invocation of `get_next()`.

To indicate that no more entries exist, we reply with a NULL pointer for the *`v`* array. The values of the *`num_vals_in_key`* and *`next`* parameters are ignored in this case.

Passing the value `-1` for *`next`* has a special meaning. It tells ConfD that we want the next request for this list traversal to use the `find_next()` (or `find_next_object()`) callback instead of `get_next()` (or `get_next_object()`).

Note

In the case of list traversal by means of a secondary index, the secondary index values must be unique for entry-by-entry traversal with `find_next()`/`find_next_object()` to be possible. Thus we can not pass `-1` for the *`next`* parameter in this case if the secondary index values are not unique.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_not_found`**`(struct confd_trans_ctx *`*`tctx`*`);`

This function is used by the `get_elem()` and `exists_optional()` callbacks to indicate to ConfD that a list entry or node does not exist.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`confd_data_reply_found`**`(struct confd_trans_ctx *`*`tctx`*`);`

This function is used by the `exists_optional()` callback to indicate to ConfD that a node does exist.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`confd_data_reply_next_object_array`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_value_t *`*`v`*`, int `*`n`*`, long `*`next`*`);`

This function is used by the optional `get_next_object()` and `find_next_object()` callbacks to return an entire object including its keys, as well as the *`next`* parameter that has the same function as for `confd_data_reply_next_key()`. It combines the functions of `confd_data_reply_next_key()` and `confd_data_reply_value_array()`.

The array of confd_value_t elements must be populated in exactly the same manner as for `confd_data_reply_value_array()` and the *`long next`* is used in the same manner as the equivalent *`next`* parameter in `confd_data_reply_next_key()`. To indicate the end of the list we - similar to `confd_data_reply_next_key()` - pass a NULL pointer for the value array.

If we are replying to a `get_next_object()` or `find_next_object()` request for an operational data list without keys (see the Operational Data chapter in the User Guide), we must include the "pseudo" key in the array, as the first element (i.e. preceding the actual leafs from the data model).

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_next_object_tag_value_array`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_tag_value_t *`*`tv`*`, int `*`n`*`, long `*`next`*`);`

This function is used by the optional `get_next_object()` and `find_next_object()` callbacks to return an entire object including its keys, as well as the *`next`* parameter that has the same function as for `confd_data_reply_next_key()`. It combines the functions of `confd_data_reply_next_key()` and `confd_data_reply_tag_value_array()`.

Similar to how the `confd_data_reply_value_array()` has its companion function `confd_data_reply_tag_value_array()` if we want to return an object as an array of confd_tag_value_t values instead of an array of confd_value_t values, we can use this function instead of `confd_data_reply_next_object_array()` when we wish to return values from the `get_next_object()` callback.

The array of confd_tag_value_t elements must be populated in exactly the same manner as for `confd_data_reply_tag_value_array()` (except that the key values must be included), and the *`long next`* is used in the same manner as the equivalent *`next`* parameter in `confd_data_reply_next_key()`. The key leafs must always be given as the first elements of the array, and in the order specified in the data model. To indicate the end of the list we - similar to `confd_data_reply_next_key()` - pass a NULL pointer for the value array.

If we are replying to a `get_next_object()` or `find_next_object()` request for an operational data list without keys (see the Operational Data chapter in the User Guide), the "pseudo" key must be included, as the first element in the array, with a tag value of 0 - i.e. it can be set with code like this:

```
confd_tag_value_t tv[7];

CONFD_SET_TAG_INT64(&tv[0], 0, 42);
```

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_next_object_arrays`**`(struct confd_trans_ctx *`*`tctx`*`, const struct confd_next_object *`*`obj`*`, int `*`nobj`*`, int `*`timeout_millisecs`*`);`

This function is used by the optional `get_next_object()` and `find_next_object()` callbacks to return multiple objects including their keys, in confd_value_t form. The struct confd_next_object is defined as:

```
struct confd_next_object {
```

```
    confd_value_t *v;
    int n;
    long next;
};
```

I.e. it corresponds exactly to the data provided for a call of `confd_data_reply_next_object_array()`. The parameter *`obj`* is a pointer to an *`nobj`* elements long array of such structs. We can also pass a timeout value for ConfD's caching of the returned data via *`timeout_millisecs`*. If we pass 0 for this parameter, the value configured via `/confdConfig/capi/objectCacheTimeout` in `confd.conf` (see confd.conf(5)) will be used.

The cache in ConfD may become invalid (e.g. due to timeout) before all the returned list entries have been used, and ConfD may then need to issue a new callback request based on an "intermediate" `next` value. This is done exactly as for the single-entry case, i.e. if `next` is `-1`, `find_next_object()` (or `find_next()`) will be used, with the keys from the "previous" entry, otherwise `get_next_object()` (or `get_next()`) will be used, with the given `next` value.

Thus a data provider can choose to give `next` values that uniquely identify list entries if that is convenient, or otherwise use `-1` for all `next` elements - or a combination, e.g. `-1` for all but the last entry. If any `next` value is given as `-1`, at least one of the `find_next()` and `find_next_object()` callbacks must be registered.

To indicate the end of the list we can either pass a NULL pointer for the *`obj`* array, or pass an array where the last struct confd_next_object element has the `v` element set to NULL. The latter is preferable, since we can then combine the final list entries with the end-of-list indication in the reply to a single callback invocation.

Note

When `next` values other than `-1` are used, these must remain valid even after the end of the list has been reached, since ConfD may still need to issue a new callback request based on an "intermediate" `next` value as described above. They can be discarded (e.g. allocated memory released) when a new `get_next_object()` or `find_next_object()` callback request for the same list in the same transaction has been received, or at the end of the transaction.

Note

In the case of list traversal by means of a secondary index, the secondary index values must be unique for entry-by-entry traversal with `find_next_object()`/`find_next()` to be possible. Thus we can not use `-1` for the `next` element in this case if the secondary index values are not unique.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_next_object_tag_value_arrays`**`(struct confd_trans_ctx *`*`tctx`*`, const struct confd_tag_next_object *`*`tobj`*`, int `*`nobj`*`, int `*`timeout_millisecs`*`);`

This function is used by the optional `get_next_object()` and `find_next_object()` callbacks to return multiple objects including their keys, in confd_tag_value_t form. The struct confd_tag_next_object is defined as:

```
struct confd_tag_next_object {
    confd_tag_value_t *tv;
    int n;
    long next;
```

```
};
```

I.e. it corresponds exactly to the data provided for a call of `confd_data_reply_next_object_tag_value_array()`. The parameter *`tobj`* is a pointer to an *`nobj`* elements long array of such structs. We can also pass a timeout value for ConfD's caching of the returned data via *`timeout_millisecs`*. If we pass 0 for this parameter, the value configured via `/confdConfig/capi/objectCacheTimeout` in `confd.conf` (see confd.conf(5)) will be used.

The cache in ConfD may become invalid (e.g. due to timeout) before all the returned list entries have been used, and ConfD may then need to issue a new callback request based on an "intermediate" `next` value. This is done exactly as for the single-entry case, i.e. if `next` is `-1`, `find_next_object()` (or `find_next()`) will be used, with the keys from the "previous" entry, otherwise `get_next_object()` (or `get_next()`) will be used, with the given `next` value.

Thus a data provider can choose to give `next` values that uniquely identify list entries if that is convenient, or otherwise use `-1` for all `next` elements - or a combination, e.g. `-1` for all but the last entry. If any `next` value is given as `-1`, at least one of the `find_next()` and `find_next_object()` callbacks must be registered.

To indicate the end of the list we can either pass a NULL pointer for the *`tobj`* array, or pass an array where the last struct confd_tag_next_object element has the `tv` element set to NULL. The latter is preferable, since we can then combine the final list entries with the end-of-list indication in the reply to a single callback invocation.

Note

When `next` values other than `-1` are used, these must remain valid even after the end of the list has been reached, since ConfD may still need to issue a new callback request based on an "intermediate" `next` value as described above. They can be discarded (e.g. allocated memory released) when a new `get_next_object()` or `find_next_object()` callback request for the same list in the same transaction has been received, or at the end of the transaction.

Note

In the case of list traversal by means of a secondary index, the secondary index values must be unique for entry-by-entry traversal with `find_next_object()`/`find_next()` to be possible. Thus we can not use `-1` for the `next` element in this case if the secondary index values are not unique.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_data_reply_attrs`**`(struct confd_trans_ctx *`*`tctx`*`, const confd_attr_value_t *`*`attrs`*`, int `*`num_attrs`*`);`

This function is used by the `get_attrs()` callback to return the requested attribute values. The *`attrs`* array should be populated with *`num_attrs`* elements of type confd_attr_value_t, which is defined as:

```
typedef struct confd_attr_value {
    u_int32_t attr;
    confd_value_t v;
} confd_attr_value_t;
```

If multiple attributes were requested in the callback invocation, they should be given in the same order in the reply as in the request. Requested attributes that are not set should be omitted from the array. If none of the requested attributes are set, or no attributes at all are set when all attributes are requested, *`num_attrs`* should be given as 0, and the value of *`attrs`* is ignored.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

`int` **`confd_delayed_reply_ok`**`(struct confd_trans_ctx *`*`tctx`*`);`

This function must be used to return the equivalent of CONFD_OK when the actual callback returned CONFD_DELAYED_RESPONSE. I.e. it is appropriate for a transaction callback, a data callback for a write operation, or a validation callback, when the result is successful.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`confd_delayed_reply_error`**`(struct confd_trans_ctx *`*`tctx`*`, const char *`*`errstr`*`);`

This function must be used to return an error when the actual callback returned CONFD_DELAYED_RESPONSE. There are two cases where the value of *`errstr`* has a special significance:

"locked" after invocation of `trans_lock()`	This is equivalent to returning CONFD_ALREADY_LOCKED from the callback.
"in_use" after invocation of `write_start()` or `prepare()`	This is equivalent to returning CONFD_IN_USE from the callback.

In all other cases, calling `confd_delayed_reply_error()` is equivalent to calling `confd_trans_seterr()` with the *`errstr`* value and returning CONFD_ERR from the callback. It is also possible to first call `confd_trans_seterr()` (for the varargs format) or `confd_trans_seterr_extended()` etc (for EXTENDED ERROR REPORTING as described in confd_lib_lib(3)), and then call `confd_delayed_reply_error()` with NULL for *`errstr`*.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`confd_data_set_timeout`**`(struct confd_trans_ctx *`*`tctx`*`, int `*`timeout_secs`*`);`

A data callback should normally complete "quickly", since e.g. the execution of a 'show' command in the CLI may require many data callback invocations. Thus it should be possible to set the `/confdConfig/capi/queryTimeout` in `confd.conf` (see above) such that it covers the longest possible execution time for any data callback. In some rare cases it may still be necessary for a data callback to have a longer execution time, and then this function can be used to extend (or shorten) the timeout for the current callback invocation. The timeout is given in seconds from the point in time when the function is called.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`void` **`confd_trans_seterr`**`(struct confd_trans_ctx *`*`tctx`*`, const char *`*`fmt`*`, ...);`

This function is used by the application to set an error string. The next transaction or data callback which returns CONFD_ERR will have this error description attached to it. This error may propagate to the CLI, the NETCONF manager, the Web UI or the log files depending on the situation. We also use this function to propagate warning messages from the `validate()` callback if we are doing semantic validation in C. The *`fmt`* argument is a printf style format string.

`void` **`confd_trans_seterr_extended`**`(struct confd_trans_ctx *`*`tctx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide more structured error information from a transaction or data callback, see the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`int` **`confd_trans_seterr_extended_info`**`(struct confd_trans_ctx *`*`tctx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, confd_tag_value_t *`*`error_info`*`, int `*`n`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide structured error information in the same way as `confd_trans_seterr_extended()`, and additionally provide contents for the NETCONF <error-info> element. See the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`void` **`confd_db_seterr`**`(struct confd_db_ctx *`*`dbx`*`, const char *`*`fmt`*`, ...);`

This function is used by the application to set an error string. The next db callback function which returns CONFD_ERR will have this error description attached to it. This error may propagate to the CLI, the NETCONF manager, the Web UI or the log files depending on the situation. The *`fmt`* argument is a printf style format string.

`void` **`confd_db_seterr_extended`**`(struct confd_db_ctx *`*`dbx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide more structured error information from a db callback, see the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`int` **`confd_db_seterr_extended_info`**`(struct confd_db_ctx *`*`dbx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, confd_tag_value_t *`*`error_info`*`, int `*`n`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide structured error information in the same way as `confd_db_seterr_extended()`, and additionally provide contents for the NETCONF <error-info> element. See the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`int` **`confd_db_set_timeout`**`(struct confd_db_ctx *`*`dbx`*`, int `*`timeout_secs`*`);`

Some of the DB callbacks registered via `confd_register_db_cb()`, e.g. `copy_running_to_startup()`, may require a longer execution time than others, and in these cases the timeout specified for `/confdConfig/capi/newSessionTimeout` may be insufficient. This function can then be used to extend the timeout for the current callback invocation. The timeout is given in seconds from the point in time when the function is called.

`int` **`confd_aaa_reload`**`(const struct confd_trans_ctx *`*`tctx`*`);`

When the ConfD AAA tree is populated by an external data provider (see the AAA chapter in the User Guide), this function can be used by the data provider to notify ConfD when there is a change to the AAA data. I.e. it is an alternative to executing the command **confd --clear-aaa-cache**. See also `maapi_aaa_reload()` in confd_lib_maapi(3).

`int` **`confd_install_crypto_keys`**`(struct confd_daemon_ctx* `*`dtx`*`);`

It is possible to define DES3 and AES keys inside confd.conf. These keys are used by ConfD to encrypt data which is entered into the system which has either of the two builtin types tailf:des3-cbc-encrypted-string or tailf:aes-cfb-128-encrypted-string. See confd_types(3).

This function will copy those keys from ConfD (which reads confd.conf) into memory in the library. The parameter *`dtx`* is a daemon context which is connected through a call to `confd_connect()`.

Note

The function must be called before `confd_register_done()` is called. If this is impractical, or if the application doesn't otherwise use a daemon context, the equivalent function `maapi_install_crypto_keys()` may be more convenient to use, see confd_lib_maapi(3).

NCS SERVICE CALLBACKS

NCS service callbacks are invoked in a manner similar to the data callbacks described above, but require a registration for a service point, specified as `ncs:servicepoint` in the data model. The `init()` transaction callback must also be registered, and must use the `confd_trans_set_fd()` function to assign a worker socket for the transaction.

`int` **`ncs_register_service_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct ncs_service_cbs *`*`scb`*`);`

This function registers the service callbacks. The struct ncs_service_cbs is defined as:

```
struct ncs_name_value {
    char *name;
    char *value;
};
```

```
enum ncs_service_operation {
    NCS_SERVICE_CREATE = 0,
    NCS_SERVICE_UPDATE = 1,
    NCS_SERVICE_DELETE = 2
};
```

```
struct ncs_service_cbs {
    char servicepoint[MAX_CALLPOINT_LEN];

    int (*pre_modification)(struct confd_trans_ctx *tctx,
                            enum ncs_service_operation op,
                            confd_hkeypath_t *kp,
                            struct ncs_name_value *proplist,
                            int num_props);
    int (*pre_lock_create)(struct confd_trans_ctx *tctx,
                           confd_hkeypath_t *kp,
                           struct ncs_name_value *proplist,
                           int num_props, int fastmap_thandle);
    int (*create)(struct confd_trans_ctx *tctx, confd_hkeypath_t *kp,
                  struct ncs_name_value *proplist, int num_props,
                  int fastmap_thandle);
    int (*post_modification)(struct confd_trans_ctx *tctx,
                             enum ncs_service_operation op,
                             confd_hkeypath_t *kp,
                             struct ncs_name_value *proplist,
                             int num_props);
    void *cb_opaque; /* private user data    */
};
```

The `create()` callback is invoked inside NCS FASTMAP when creation or update of a service instance is committed. It should attach to the FASTMAP transaction by means of `maapi_attach2()` (see confd_lib_maapi(3)), passing the *`fastmap_thandle`* transaction handle as the *`thandle`* parameter to `maapi_attach2()`. The *`usid`* parameter for `maapi_attach2()` should be given as 0. To modify

data in the FASTMAP transaction, the NCS-specific `maapi_shared_xxx()` functions must be used, see the section NCS SPECIFIC FUNCTIONS in the confd_lib_maapi(3) manual page.

The `pre_lock_create()` callback is invoked in the same way as the `create()` callback. The difference is that this callback is invoked outside the transaction lock of the current transaction, and may thus run in parallel with `pre_lock_create()` invocations in other transactions. Typically a service will only register one or the other of `create()` and `pre_lock_create()`, but it is possible to register both if there is a part of the service creation that can be performed in parallel with other invocation and a part that has to be protected by the transaction lock to avoid conflicts.

The `pre_modification()` and `post_modification()` callbacks are optional, and are invoked outside FASTMAP. `pre_modification()` is invoked before create, update, or delete of the service, as indicated by the *`enum ncs_service_operation op`* parameter. Conversely `post_modification()` is invoked after create, update, or delete of the service. These functions can be useful e.g. for allocations that should be stored and existing also when the service instance is removed.

All the callbacks receive a property list via the *`proplist`* and *`num_props`* parameters. This list is initially empty (*`proplist`* == NULL and *`num_props`* == 0), but it can be used to store and later modify persistent data outside the service model that might be needed.

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

`int` **`ncs_service_reply_proplist`**`(struct confd_trans_ctx *`*`tctx`*`, const struct ncs_name_value *`*`proplist`*`, int `*`num_props`*`);`

This function must be called with the new property list, immediately prior to returning from the callback, if the stored property list should be updated. If a callback returns without calling `ncs_service_reply_proplist()`, the previous property list is retained. To completely delete the property list, call this function with the *`num_props`* parameter given as 0.

VALIDATION CALLBACKS

This library also supports the registration of callback functions on validation points in the data model. A validation point is a point in the data model where ConfD will invoke an external function to validate the associated data. The validation occurs before a transaction is committed. Similar to the state machine described for "external data bases" above where we install callback functions in the struct confd_trans_cbs, we have to install callback functions for each validation point. It does not matter if the database is CDB or an external database, the validation callbacks described here work equally well for both cases.

`void` **`confd_register_trans_validate_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_trans_validate_cbs *`*`vcbs`*`);`

This function installs two callback functions for the struct confd_daemon_ctx. One function that gets called when the validation phase starts in a transaction and one when the validation phase stops in a transaction. In the `init()` callback we can use the MAAPI api to attach to the running transaction, this way we can later on, freely traverse the configuration and read data. The data we will be reading through MAAPI (see confd_lib_maapi(3)) will be read from the shadow storage containing the *not-yet-committed* data.

The struct confd_trans_validate_cbs is defined as:

```
struct confd_trans_validate_cbs {
    int (*init)(struct confd_trans_ctx *tctx);
    int (*stop)(struct confd_trans_ctx *tctx);
```

```
};
```

It must thus be populated with two function pointers when we call this function.

The `init()` callback is conceptually invoked at the start of the validation phase, but just as for transaction callbacks, ConfD will as far as possible delay the actual invocation of the validation `init()` callback for a given daemon until it is required. This means that if none of the daemon's `validate()` callbacks need to be invoked (see below), `init()` and `stop()` will not be invoked either.

If we need to allocate memory or other resources for the validation this can also be done in the `init()` callback, with the resources being freed in the `stop()` callback. We can use the `t_opaque` element in the struct confd_trans_ctx to manage this, but in a daemon that implements both data and validation callbacks it is better to use the `v_opaque` element for validation, to be able to manage the allocations independently.

Similar to the `init()` callback for external data bases, we must in the `init()` callback associate a file descriptor with the transaction. This file descriptor will be used for the actual validation. Thus in a multi threaded application, we can have one thread performing validation for a transaction in parallel with other threads executing e.g. data callbacks. Thus a typical implementation of an `init()` callback for validation looks as:

```
static int init_validation(struct confd_trans_ctx *tctx)
{
    maapi_attach(maapi_socket, mtest__ns, tctx);
    confd_trans_set_fd(tctx, workersock);
    return CONFD_OK;
}
```

`int` **`confd_register_valpoint_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_valpoint_cb *`*`vcb`*`);`

We must also install an actual validation function for each validation point, i.e. for each `tailf:validate` statement in the YANG data model.

A validation point has a name and an associated function pointer. The struct which must be populated for each validation point looks like:

```
struct confd_valpoint_cb {
    char valpoint[MAX_CALLPOINT_LEN];
    int (*validate)(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *kp,
                    confd_value_t *newval);
    void *cb_opaque;        /* private user data */
};
```

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

See the user guide chapter "Semantic validation" for code examples. The `validate()` callback can return CONFD_OK if all is well, or CONFD_ERROR if the validation fails. If we wish a message to accompany the error we must prior to returning from the callback, call `confd_trans_seterr()` or `confd_trans_seterr_extended()`.

The `cb_opaque` element can be used to pass arbitrary data to the callback, e.g. when the same callback is used for multiple validation points. It is made available to the callback via the element `vcb_opaque` in the transaction context (*`tctx`* argument), see the structure definition above.

If the `tailf:opaque` substatement has been used with the `tailf:validate` statement in the data model, the argument string is made available to the callback via the `validate_opaque` element in the transaction context.

We also have yet another special return value which can be used (only) from the `validate()` callback which is CONFD_VALIDATION_WARN. Prior to return of this value we must call `confd_trans_seterr()` which provides a string describing the warning. The warnings will get propagated to the transaction engine, and depending on where the transaction originates, ConfD may or may not act on the warnings. If the transaction originates from the CLI or the Web UI, ConfD will interactively present the user with a choice - whereby the transaction can be aborted.

If the transaction originates from NETCONF - which does not have any interactive capabilities - the warnings are ignored. The warnings are primarily intended to alert inexperienced users that attempt to make - dangerous - configuration changes. There can be multiple warnings from multiple validation points in the same transaction.

It is also possible to let the `validate()` callback return CONFD_DELAYED_RESPONSE in which case the application at a later stage must invoke either `confd_delayed_reply_ok()`, `confd_delayed_reply_error()` or `confd_delayed_reply_validation_warn()`.

In some cases it may be necessary for the validation callbacks to verify the availability of resources that will be needed if the new configuration is committed. To support this kind of verification, the `validation_info` element in the struct confd_trans_ctx can carry one of these flags:

CONFD_VALIDATION_FLAG_TEST
: When this flag is set, the current validation phase is a "test" validation, as in e.g. the CLI 'validate' command, and the transaction will return to the READ state regardless of the validation result. This flag is available in all of the `init()`, `validate()`, and `stop()` callbacks.

CONFD_VALIDATION_FLAG_COMMIT
: When this flag is set, all requirements for a commit have been met, i.e. all validation as well as the write_start and prepare transitions have been successful, and the actual commit will follow. This flag is only available in the `stop()` callback.

`int` **`confd_register_range_valpoint_cb`**`(struct confd_daemon_ctx *`*`dx`*`, struct confd_valpoint_cb *`*`vcb`*`, const confd_value_t *`*`lower`*`, const confd_value_t *`*`upper`*`, int `*`numkeys`*`, const char *`*`fmt`*`, ...);`

A variant of `confd_register_valpoint_cb()` which registers a validation function for a range of key values. The *`lower`*, *`upper`*, *`numkeys`*, *`fmt`*, and remaining parameters are the same as for `confd_register_range_data_cb()`, see above.

`int` **`confd_delayed_reply_validation_warn`**`(struct confd_trans_ctx *`*`tctx`*`);`

This function must be used to return the equivalent of CONFD_VALIDATION_WARN when the `validate()` callback returned CONFD_DELAYED_RESPONSE. Before calling this function, we must call `confd_trans_seterr()` to provide a string describing the warning.

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS

NOTIFICATION STREAMS

The application can generate notifications that are sent via the northbound protocols. Currently NETCONF notification streams are supported. The application generates the content for each notification and sends it via a socket to ConfD, which in turn manages the stream subscriptions and distributes the notifications accordingly.

A stream always has a "live feed", which is the sequence of new notifications, sent in real time as they are generated. Subscribers may also request "replay" of older, logged notifications if the stream supports this, perhaps transitioning to the live feed when the end of the log is reached. There may be one or more replays active simultaneously with the live feed. ConfD forwards replay requests from subscribers to the application via callbacks if the stream supports replay.

Each notification has an associated time stamp, the "event time". This is the time when the event that generated the notification occurred, rather than the time the notification is logged or sent, in case these times differ. The application must pass the event time to ConfD when sending a notification, and it is also needed when replaying logged events, see below.

`int` **`confd_register_notification_stream`**`(struct confd_daemon_ctx *dx, const struct confd_notification_stream_cbs *ncbs, struct confd_notification_ctx **nctx);`

This function registers the notification stream and optionally two callback functions used for the replay functionality. If the stream does not support replay, the callback elements in the struct confd_notification_stream_cbs are set to NULL. A context pointer is returned via the `**nctx` argument - this must be used by the application for the sending of live notifications via `confd_notification_send()` (see below).

The confd_notification_stream_cbs structure is defined as:

```
struct confd_notification_stream_cbs {
    char streamname[MAX_STREAMNAME_LEN];
    int fd;
    int (*get_log_times)(
        struct confd_notification_ctx *nctx);
    int (*replay)(struct confd_notification_ctx *nctx,
                  struct confd_datetime *start,
                  struct confd_datetime *stop);
    void *cb_opaque;        /* private user data */
};
```

The `fd` element must be set to a previously connected worker socket. This socket may be used for multiple notification streams, but not for any of the callback processing described above. Since it is only used for sending data to ConfD, there is no need for the application to poll the socket. Note that the control socket must be connected before registration even if the callbacks are not registered.

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

The `get_log_times()` callback is called by ConfD to find out a) the creation time of the current log and b) the event time of the last notification aged out of the log, if any. The application provides the times via the `confd_notification_reply_log_times()` function (see below) and returns CONFD_OK.

The `replay()` callback is called by ConfD to request replay. The `nctx` context pointer must be saved by the application and used when sending the replay notifications via `confd_notification_send()`, as well as for the `confd_notification_replay_complete()` (or `confd_notification_replay_failed()`) call (see below) - the callback should return without waiting for the replay to complete. The pointer references allocated memory, which is freed by the `confd_notification_replay_complete()` (or `confd_notification_replay_failed()`) call.

The times given by *`*start`* and *`*stop`* specify the extent of the replay. The start time will always be given and specify a time in the past, however the stop time may be either in the past or in the future or even omitted, i.e. the *`stop`* argument is NULL. This means that the subscriber has requested that the subscription continues indefinitely with the live feed when the logged notifications have been sent.

If the stop time is given:

- The application sends all logged notifications that have an event time later than the start time but not later than the stop time, and then calls `confd_notification_replay_complete()`. Note that if the stop time is in the future when the replay request arrives, this includes notifications logged while the replay is in progress (if any), as long as their event time is not later than the stop time.

If the stop time is *not* given:

- The application sends all logged notifications that have an event time later than the start time, and then calls `confd_notification_replay_complete()`. Note that this includes notifications logged after the request was received (if any).

ConfD will if needed switch the subscriber over to the live feed and then end the subscription when the stop time is reached. The callback may analyze the *`start`* and *`stop`* arguments to determine start and stop positions in the log, but if the analysis is postponed until after the callback has returned, the confd_datetime structure(s) must be copied by the callback.

The `replay()` callback may optionally select a separate worker socket to be used for the replay notifications. In this case it must call `confd_notification_set_fd()` to indicate which socket should be used.

Note that unlike the callbacks for external data bases and validation, these callbacks do not use a worker socket for the callback processing, and consequently there is no `init()` callback to request one. The callbacks are invoked, and the reply is sent, via the daemon control socket.

The `cb_opaque` element in the confd_notification_stream_cbs structure can be used to pass arbitrary data to the callbacks in much the same way as for callpoint and validation point registrations, see the description of the struct confd_data_cbs structure above. However since the callbacks are not associated with a transaction, this element is instead made available in the confd_notification_ctx structure.

`int` **`confd_notification_send`**`(struct confd_notification_ctx *`*`nctx`*`, struct confd_datetime *`*`time`*`, confd_tag_value_t *`*`values`*`, int `*`nvalues`*`);`

This function is called by the application to send a notification, whether "live" or replay. The *`nctx`* pointer is provided by ConfD as described above. The *`time`* argument specifies the event time for the notification. The *`values`* argument is an array of length *`nvalues`*, populated with the content of the notification as described for the Tagged Value Array format in the XML STRUCTURES section of the confd_types(3) manual page.

For example, a NETCONF notification of the form

```
<ncn:notification
 xmlns:ncn="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <linkUp xmlns="http://example.com/ns/test/1.0">
    <ncn:eventTime>2007-08-17T08:56:05Z</ncn:eventTime>
    <ifIndex>3</ifIndex>
  </linkUp>
</ncn:notification>
```

could be sent with the following code:

```
struct confd_notification_ctx *nctx;
struct confd_datetime event_time = {2007, 8, 17, 8, 56, 5, 0, 0, 0};
confd_tag_value_t notif[3];
int n = 0;

CONFD_SET_TAG_XMLBEGIN(&notif[n], test_linkUp, test__ns); n++;
CONFD_SET_TAG_UINT32(&notif[n], test_ifIndex, 3); n++;
CONFD_SET_TAG_XMLEND(&notif[n], test_linkUp, test__ns); n++;
confd_notification_send(nctx, &event_time, notif, n);
```

Note

While it is possible to use separate threads to send live and replay notifications for a given stream, or to send different streams on a given worker socket, this is not recommended. This is because it involves rather complex synchronization problems that can only be fully solved by the application, in particular in the case where a replay switches over to the live feed.

`int` **`confd_notification_replay_complete`**`(struct confd_notification_ctx *`*`nctx`*`);`

The application calls this function to notify ConfD that the replay is complete, using the *`nctx`* pointer received in the corresponding `replay()` callback invocation.

`int` **`confd_notification_replay_failed`**`(struct confd_notification_ctx *`*`nctx`*`);`

In case the application fails to complete the replay as requested (e.g. the log gets overwritten while the replay is in progress), the application should call this function *instead* of `confd_notification_replay_complete()`. An error message describing the reason for the failure can be supplied by first calling `confd_notification_seterr()` or `confd_notification_seterr_extended()`, see below. The *`nctx`* pointer received in the corresponding `replay()` callback invocation is used for both calls.

`void` **`confd_notification_set_fd`**`(struct confd_notification_ctx *`*`nctx`*`, int` *`fd`*`);`

This function may optionally be called by the `replay()` callback to request that the worker socket given by *`fd`* should be used for the replay. Otherwise the socket specified in the confd_notification_stream_cbs at registration will be used.

`int` **`confd_notification_reply_log_times`**`(struct confd_notification_ctx *`*`nctx`*`, struct confd_datetime *`*`creation`*`, struct confd_datetime *`*`aged`*`);`

Reply function for use in the `get_log_times()` callback invocation. If no notifications have been aged out of the log, give NULL for the *`aged`* argument.

`void` **`confd_notification_seterr`**`(struct confd_notification_ctx *`*`nctx`*`, const char *`*`fmt`*`, ...);`

In some cases the callbacks may be unable to carry out the requested actions, e.g. the capacity for simultaneous replays might be exceeded, and they can then return CONFD_ERR. This function allows the callback to associate an error message with the failure. It can also be used to supply an error message before calling `confd_notification_replay_failed()`.

`void` **`confd_notification_seterr_extended`**`(struct confd_notification_ctx *`*`nctx`*`, enum confd_errcode` *`code`*`, u_int32_t` *`apptag_ns`*`, u_int32_t` *`apptag_tag`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide more structured error information from a notification callback, see the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`int` **`confd_notification_seterr_extended_info`**`(struct confd_notification_ctx *nctx, enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, confd_tag_value_t *error_info, int n, const char *fmt, ...);`

This function can be used to provide structured error information in the same way as `confd_notification_seterr_extended()`, and additionally provide contents for the NETCONF <error-info> element. See the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

`int` **`confd_register_snmp_notification`**`(struct confd_daemon_ctx *dx, int fd, const char *notify_name, const char *ctx_name, struct confd_notification_ctx **nctx);`

SNMP notifications can also be sent via the notification framework, however most aspects of the stream concept described above do not apply for SNMP. This function is used to register a worker socket, the snmpNotifyName (*`notify_name`*), and SNMP context (*`ctx_name`*) to be used for the notifications.

The *`fd`* parameter must give a previously connected worker socket. This socket may be used for different notifications, but not for any of the callback processing described above. Since it is only used for sending data to ConfD, there is no need for the application to poll the socket. Note that the control socket must be connected before registration, even if none of the callbacks described below are registered.

The context pointer returned via the *`**nctx`* argument must be used by the application for the subsequent sending of the notifications via `confd_notification_send_snmp()` or `confd_notification_send_snmp_inform()` (see below).

When a notification is sent using one of these functions, it is delivered to the management targets defined for the `snmpNotifyName` in the `snmpNotifyTable` in SNMP-NOTIFICATION-MIB for the specified SNMP context. If *`notify_name`* is NULL or the empty string (""), the notification is sent to all management targets. If *`ctx_name`* is NULL or the empty string (""), the default context ("") is used.

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

`int` **`confd_notification_send_snmp`**`(struct confd_notification_ctx *nctx, const char *notification, struct confd_snmp_varbind *varbinds, int num_vars);`

Sends the SNMP notification specified by *`notification`*, without requesting inform-request delivery information. This is equivalent to calling `confd_notification_send_snmp_inform()` (see below) with NULL as the *`cb_id`* argument. I.e. if the common arguments are the same, the two functions will send the exact same set of traps and inform-requests.

`int` **`confd_register_notification_snmp_inform_cb`**`(struct confd_daemon_ctx *dx, const struct confd_notification_snmp_inform_cbs *cb);`

If we want to receive information about the delivery of SNMP inform-requests, we must register two callbacks for this. The struct confd_notification_snmp_inform_cbs is defined as:

```
struct confd_notification_snmp_inform_cbs {
    char cb_id[MAX_CALLPOINT_LEN];
```

```
    void (*targets)(struct confd_notification_ctx *nctx,
                    int ref, struct confd_snmp_target *targets,
                    int num_targets);
    void (*result)(struct confd_notification_ctx *nctx,
                   int ref, struct confd_snmp_target *target,
                   int got_response);
    void *cb_opaque;        /* private user data */
};
```

The callback identifier *cb_id* can be chosen arbitrarily, it is only used when sending SNMP notifications with confd_notification_send_snmp_inform() - however each inform callback registration must use a unique *cb_id*. The callbacks are invoked via the control socket, i.e. the application must poll it and invoke confd_fd_ready() when data is available.

When a notification is sent, the target() callback will be invoked once with *num_targets* (possibly 0) inform-request targets in the *targets* array, followed by *num_targets* invocations of the result() callback, one for each target. The *ref* argument (passed from the confd_notification_send_snmp_inform() call) allows for tracking the result of multiple notifications with delivery overlap.

Note

We must call the confd_register_done() function when we are done with all registrations for a daemon, see above.

int **confd_notification_send_snmp_inform**(struct confd_notification_ctx *nctx, const char *notification, struct confd_snmp_varbind *varbinds, int num_vars, const char *cb_id, int ref);

Sends the SNMP notification specified by *notification*. If *cb_id* is not NULL, the callbacks registered for *cb_id* will be invoked with the *ref* argument as described above, otherwise no inform-request delivery information will be provided. The *varbinds* array should be populated with *num_vars* elements as described in the Notifications section of the SNMP Agent chapter in the User Guide.

If *notification* is the empty string, no notification is looked up; instead *varbinds* defines the notification, including the notification id (variable name "snmpTrapOID"). This is especially useful for forwarding a notification which has been received from the SNMP gateway (see confd_register_notification_sub_snmp_cb() below).

If *varbinds* does not contain a timestamp (variable name "sysUpTime"), one will be supplied by the agent.

void **confd_notification_set_snmp_src_addr**(struct confd_notification_ctx *nctx, const struct confd_ip *src_addr);

By default, the source address for the SNMP notifications that are sent by the above functions is chosen by the IP stack of the OS. This function may be used to select a specific source address, given by *src_addr*, for the SNMP notifications subsequently sent using the *nctx* context. The default can be restored by calling the function with a *src_addr* where the af element is set to AF_UNSPEC.

int **confd_notification_set_snmp_notify_name**(struct confd_notification_ctx *nctx, const char *notify_name);

This function can be used to change the snmpNotifyName (*notify_name*) for the *nctx* context. The new snmpNotifyName is used for notifications sent by subsequent calls to confd_notification_send_snmp() and confd_notification_send_snmp_inform() that use the *nctx* context.

```
int confd_register_notification_sub_snmp_cb(struct confd_daemon_ctx
*dx, const struct confd_notification_sub_snmp_cb *cb);
```

Registers a callback function to be called when an SNMP notification is received by the SNMP gateway.

The struct confd_notification_sub_snmp_cb is defined as:

```
struct confd_notification_sub_snmp_cb {
    char sub_id[MAX_CALLPOINT_LEN];
    int (*recv)(struct confd_notification_ctx *nctx,
                char *notification,
                struct confd_snmp_varbind *varbinds, int num_vars,
                confd_value_t *src_addr, u_int16_t src_port);
    void *cb_opaque;        /* private user data */
};
```

The `sub_id` element is the subscription id for the notifications. The `recv()` callback will be called when a notification is received. See the section "Receiving and Forwarding Traps" in the chapter "The SNMP gateway" in the Users Guide.

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

```
int confd_notification_flush(struct confd_notification_ctx *nctx);
```

Notifications are sent asynchronously, i.e. normally without blocking the caller of the send functions described above. This means that in some cases, ConfD's sending of the notifications on the northbound interfaces may lag behind the send calls. If we want to make sure that the notifications have actually been sent out, e.g. in some shutdown procedure, we can call `confd_notification_flush()`. This function will block until all notifications sent using the given *`nctx`* context have been fully processed by ConfD. It can be used both for notification streams and for SNMP notifications (however it will not wait for replies to SNMP inform-requests to arrive).

CONFD ACTIONS

The use of action callbacks can be specified either via a `rpc` statement or via a `tailf:action` statement in the YANG data model, see the YANG specification and tailf_yang_extensions(5). In both cases the use of a `tailf:actionpoint` statement specifies that the action is implemented as a callback function. This section describes how such callback functions should be implemented and registered with ConfD.

Unlike the callbacks for data and validation, there is not always a transaction associated with an action callback. However an action is always associated with a user session (NETCONF, CLI, etc), and only one action at a time can be invoked from a given user session. Hence a pointer to the associated struct confd_user_info is passed to the callbacks.

The action callback mechanism is also used for command and completion callbacks configured for the CLI, either in a YANG module using tailf extension statements, or in a clispec(5). As the parameter structure is significantly different, special callbacks are used for these functions.

```
int confd_register_action_cbs(struct confd_daemon_ctx *dx, const struct
confd_action_cbs *acb);
```

This function registers up to five callback functions, two of which will be called in sequence when an action is invoked. The struct confd_action_cbs is defined as:

```
struct confd_action_cbs {
```

```
    char actionpoint[MAX_CALLPOINT_LEN];
    int (*init)(struct confd_user_info *uinfo);
    int (*abort)(struct confd_user_info *uinfo);
    int (*action)(struct confd_user_info *uinfo,
                  struct xml_tag *name,
                  confd_hkeypath_t *kp,
                  confd_tag_value_t *params,
                  int nparams);
    int (*command)(struct confd_user_info *uinfo,
                   char *path, int argc, char **argv);
    int (*completion)(struct confd_user_info *uinfo,
                      int cli_style, char *token, int completion_char,
                      confd_hkeypath_t *kp,
                      char *cmdpath, char *cmdparam_id,
                      struct confd_qname *simpleType, char *extra);
    void *cb_opaque;        /* private user data */
};
```

The `init()` callback, and at least one of the `action()`, `command()`, and `completion()` callbacks, must be specified. It is in principle possible to use a single "point name" for more than one of these callback types, and have the corresponding callback invoked in each case, but in typical usage we would only register one of the callbacks `action()`, `command()`, and `completion()`. Below, the term "action callback" is used to refer to any of these three.

Similar to the `init()` callback for external data bases, we must in the `init()` callback associate a worker socket with the action. This socket will be used for the invocation of the action callback, which actually carries out the action. Thus in a multi threaded application, actions can be dispatched to different threads.

However note that unlike the callbacks for external data bases and validation, both `init()` and action callbacks are registered for each action point (i.e. different action points can have different `init()` callbacks), and there is no `finish()` callback - the action is completed when the action callback returns.

The struct confd_action_ctx actx element inside the struct confd_user_info holds action-specific data, in particular the `t_opaque` element could be used to pass data from the `init()` callback to the action callback, if needed. If the action is associated with a transaction, the `thandle` element is set to the transaction handle, and can be used with a call to `maapi_attach2()` (see confd_lib_maapi(3)). This is the case for all types of action callbacks invoked from the CLI and Web UI, and for the action() callback when invoked via `maapi_request_action_th()` (see confd_lib_maapi(3)) - in other cases, this element is -1.

The `cb_opaque` element in the confd_action_cbs structure can be used to pass arbitrary data to the callbacks in much the same way as for callpoint and validation point registrations, see the description of the struct confd_data_cbs structure above. This element is made available in the confd_action_ctx structure.

If the `tailf:opaque` substatement has been used with the `tailf:actionpoint` statement in the data model, the argument string is made available to the callbacks via the `actionpoint_opaque` element in the confd_action_ctx structure.

Note

We must call the `confd_register_done()` function when we are done with all registrations for a daemon, see above.

The `action()` callback receives all the parameters pertaining to the action: The *`name`* argument is a pointer to the action name as defined in the data model, the *`kp`* argument gives the path through the data model for an action defined via `tailf:action` (it is a NULL pointer for an action defined via `rpc`), and finally the *`params`* argument is a representation of the inout parameters provided when the action is

invoked. The *params* argument is an array of length *nparams*, populated as described for the Tagged Value Array format in the XML STRUCTURES section of the confd_types(3) manual page.

The command() callback is invoked for CLI callback commands. It must always result in a call of confd_action_reply_command(). As the parameters in this case are all in string form, they are passed in the traditional Unix *argc*, *argv* manner - i.e. *argv* is an array of *argc* pointers to NUL-terminated strings plus a final NULL pointer element, and *argv[0]* is the name of the command. Additionally the full path of the command is available via the *path* argument.

The completion() callback is invoked for CLI completion and information. It must result in a call of confd_action_reply_completion(), except for the case when the callback is invoked via a tailf:cli-custom-range-enumerator statement in the data model (see below). The *cli_style* argument gives the style of the CLI session as a character: 'J', 'C', or 'I'. The *token* argument is a NUL-terminated string giving the parameter of the CLI command line that the callback invocation pertains to, and *completion_char* is the character that the user typed, i.e. TAB ('\t'), SPACE (' '), or '?'. If the callback pertains to a data model element, *kp* identifies that element, otherwise it is NULL. The *cmdpath* is a NUL-terminated string giving the full path of the command. If a cli-completion-id is specified in the YANG module, or a completionId is specified in the clispec, it is given as a NUL-terminated string via *cmdparam_id*, otherwise this argument is NULL. If the invocation pertains to an element that has a type definition, the *simpleType* argument identifies the type with namespace and type name, otherwise it is NULL. The *extra* argument is currently unused (always NULL).

When completion() is invoked via a tailf:cli-custom-range-enumerator statement in the data model, it is a request to provide possible key values for creation of an entry in a list with a custom range specification. The callback must in this case result in a call of confd_action_reply_range_enum(). Refer to the cli/range_create example in the bundled examples collection to see an implementation of such a callback.

The action callbacks must return CONFD_OK, CONFD_ERR, or CONFD_DELAYED_RESPONSE. CONFD_DELAYED_RESPONSE implies that the application must later reply asynchronously.

The optional abort() callback is called whenever an action is aborted, e.g. when a user invokes an action from one of the northbound agents and aborts it before it has completed. The abort() callback will be invoked on the control socket. It is the responsibility of the abort() callback to make sure that the pending reply from the action callback is sent. This is required to allow the worker socket to be used for further queries. There are several possible ways for an application to support aborting. E.g. the application can return CONFD_DELAYED_RESPONSE from the action callback. Then, when the abort() callback is called, it can terminate the executing action and use e.g. confd_action_delayed_reply_error(). Alternatively an application can use threads where the action callback is executed in a separate thread. In this case the abort() callback could inform the thread executing the action that it should be terminated, and that thread can just return from the action callback.

```
int confd_register_range_action_cbs(struct confd_daemon_ctx *dx, con-
st struct confd_action_cbs *acb, const confd_value_t *lower, const
confd_value_t *upper, int numkeys, const char *fmt, ...);
```

A variant of confd_register_action_cbs() which registers action callbacks for a range of key values. The *lower*, *upper*, *numkeys*, *fmt*, and remaining parameters are the same as for confd_register_range_data_cb(), see above.

Note

This function can not be used for registration of the command() or completion() callbacks - only actions specified in the data model are invoked via a keypath that can be used for selection of the corresponding callbacks.

```
void confd_action_set_fd(struct confd_user_info *uinfo, int sock);
```

Associate a worker socket with the action. This function must be called in the `init()` callback - a typical implementation of an `init()` callback looks as:

```
static int init_action(struct confd_user_info *uinfo)
{
    confd_action_set_fd(uinfo, workersock);
    return CONFD_OK;
}
```

```
int confd_action_reply_values(struct confd_user_info *uinfo,
confd_tag_value_t *values, int nvalues);
```

If the action definition specifies that the action should return data, it must invoke this function in response to the `action()` callback. The *`values`* argument points to an array of length *`nvalues`*, populated with the output parameters in the same way as the *`params`* array above.

Note

This function must only be called for an `action()` callback.

```
int confd_action_reply_command(struct confd_user_info *uinfo, char
**values, int nvalues);
```

If a CLI callback command should return data, it must invoke this function in response to the `command()` callback. The *`values`* argument points to an array of length *`nvalues`*, populated with pointers to NUL-terminated strings.

Note

This function must only be called for a `command()` callback.

```
int confd_action_reply_rewrite(struct confd_user_info *uinfo, char
**values, int nvalues, char **unhides, int nunhides);
```

This function can be called instead of `confd_action_reply_command()` as a response to a show path rewrite callback invocation. The *`values`* argument points to an array of length *`nvalues`*, populated with pointers to NUL-terminated strings representing the tokens of the new path. The *`unhides`* argument points to an array of length *`nunhides`*, populated with pointers to NUL-terminated strings representing hide groups to temporarily unhide during evaluation of the show command.

Note

This function must only be called for a `command()` callback.

```
int confd_action_reply_rewrite2(struct confd_user_info *uinfo, char
**values, int nvalues, char **unhides, int nunhides, struct
confd_rewrite_select **selects, int nselects);
```

This function can be called instead of `confd_action_reply_command()` as a response to a show path rewrite callback invocation. The *`values`* argument points to an array of length *`nvalues`*, populated with pointers to NUL-terminated strings representing the tokens of the new path. The *`unhides`* argument points to an array of length *`nunhides`*, populated with pointers to NUL-terminated strings representing hide groups to temporarily unhide during evaluation of the show command. The *`selects`* argument points to an array of length *`nselects`*, populated with pointers to confd_rewrite_select structs representing additional select targets.

Note

This function must only be called for a `command()` callback.

`int` **`confd_action_reply_completion`**`(struct confd_user_info *`*`uinfo`*`, struct confd_completion_value *`*`values`*`, int `*`nvalues`*`);`

This function must normally be called in response to the `completion()` callback. The *`values`* argument points to an *`nvalues`* long array of confd_completion_value elements:

```
enum confd_completion_type {
    CONFD_COMPLETION,
    CONFD_COMPLETION_INFO,
    CONFD_COMPLETION_DESC,
    CONFD_COMPLETION_DEFAULT
};
```

```
struct confd_completion_value {
    enum confd_completion_type type;
    char *value;
    char *extra;
};
```

For a completion alternative, `type` is set to CONFD_COMPLETION, `value` gives the alternative as a NUL-terminated string, and `extra` gives explanatory text as a NUL-terminated string - if there is no such text, `extra` is set to NULL. For "info" or "desc" elements, `type` is set to CONFD_COMPLETION_INFO or CONFD_COMPLETION_DESC, respectively, and `value` gives the text as a NUL-terminated string (the `extra` element is ignored).

In order to fallback to the normal completion behavior, `type` should be set to CONFD_COMPLETION_DEFAULT. CONFD_COMPLETION_DEFAULT cannot be combined with the other completion types, implying the *`values`* array always must have length `1` which is indicated by *`nvalues`* setting.

Note

This function must only be called for a `completion()` callback.

`int` **`confd_action_reply_range_enum`**`(struct confd_user_info *`*`uinfo`*`, char **`*`values`*`, int `*`keysize`*`, int `*`nkeys`*`);`

This function must be called in response to the `completion()` callback when it is invoked via a `tailf:cli-custom-range-enumerator` statement in the data model. The *`values`* argument points to a *`keysize`* * *`nkeys`* long array of strings giving the possible key values, where *`keysize`* is the number of keys for the list in the data model and *`nkeys`* is the number of list entries for which keys are provided. I.e. the array gives entry1-key1, entry1-key2, ..., entry2-key1, entry2-key2, ... and so on. See the `cli/range_create` example in the bundled examples collection for details.

Note

This function must only be called for a `completion()` callback.

`void` **`confd_action_seterr`**`(struct confd_user_info *`*`uinfo`*`, const char *`*`fmt`*`, ...);`

If action callback encounters fatal problems that can not be expressed via the reply function, it may call this function with an appropriate message and return CONFD_ERR instead of CONFD_OK.

```
void confd_action_seterr_extended(struct confd_user_info *uinfo, enum
confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag, const
char *fmt, ...);
```

This function can be used to provide more structured error information from an action callback, see the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

```
int confd_action_seterr_extended_info(struct confd_user_info *uinfo,
enum confd_errcode code, u_int32_t apptag_ns, u_int32_t apptag_tag,
confd_tag_value_t *error_info, int n, const char *fmt, ...);
```

This function can be used to provide structured error information in the same way as `confd_action_seterr_extended()`, and additionally provide contents for the NETCONF <error-info> element. See the section EXTENDED ERROR REPORTING in confd_lib_lib(3).

```
int confd_action_delayed_reply_ok(struct confd_user_info *uinfo);
```

```
int confd_action_delayed_reply_error(struct confd_user_info *uinfo,
const char *errstr);
```

If we use the CONFD_DELAYED_RESPONSE as a return value from the action callback, we must later asynchronously reply. If we use one of the `confd_action_reply_xxx()` functions, this is a complete reply. Otherwise we must use the `confd_action_delayed_reply_ok()` function to signal success, or the `confd_action_delayed_reply_error()` function to signal an error.

```
int confd_action_set_timeout(struct confd_user_info *uinfo, int
timeout_secs);
```

Some action callbacks may require a significantly longer execution time than others, and this time may not even be possible to determine statically (e.g. a file download). In such cases the `/confdConfig/capi/queryTimeout` setting in `confd.conf` (see above) may be insufficient, and this function can be used to extend (or shorten) the timeout for the current callback invocation. The timeout is given in seconds from the point in time when the function is called.

Examples on how to work with actions are available in the User Guide and in the bundled examples collection.

AUTHENTICATION CALLBACK

We can register a callback with ConfD's AAA subsystem, to be invoked whenever AAA has completed processing of an authentication attempt. In the case where the authentication was otherwise successful, the callback can still cause it to be rejected. This can be used to implement specific access policies, as an alternative to using PAM or "External" authentication for this purpose. The callback will only be invoked if it is both enabled via `/confdConfig/aaa/authenticationCallback/enabled` in `confd.conf` (see confd.conf(5)) and registered as described here.

Note

If the callback is enabled in `confd.conf` but not registered, or invocation keeps failing for some reason, *all* authentication attempts will fail.

Note

This callback can not be used to actually *perform* the authentication. If we want to implement the authentication outside of ConfD, we need to use PAM or "External" authentication, see the AAA chapter in the User Guide.

`int` **`confd_register_auth_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_auth_cb *`*`acb`*`);`

Registers the authentication callback. The struct confd_auth_cb is defined as:

```
struct confd_auth_cb {
    int (*auth)(struct confd_auth_ctx *actx);
};
```

The `auth()` callback is invoked with a pointer to an authentication context that provides information about the result of the authentication so far. The callback must return CONFD_OK or CONFD_ERR, see below. The struct confd_auth_ctx is defined as:

```
struct confd_auth_ctx {
    struct confd_user_info *uinfo;
    char *method;
    int success;
    union {
        struct {         /* if success */
            int ngroups;
            char **groups;
        } succ;
        struct {         /* if !success */
            int logno;   /* number from confd_logsyms.h */
            char *reason;
        } fail;
    } ainfo;
    /* ConfD internal fields */
    char *errstr;
};
```

The `uinfo` element points to a struct confd_user_info with details about the user logging in, specifically user name, password (if used), source IP address, context, and protocol. Note that the user session does not actually exist at this point, even if the AAA authentication was successful - it will only be created if the callback accepts the authentication, hence e.g. the `usid` element is always 0.

The `method` string gives the authentication method used, as follows:

"password"	Password authentication. This generic term is used if the authentication failed.
"local", "pam", "external"	Password authentication. On successful authentication, the specific method that succeeded is given. See the AAA chapter in the User Guide for an explanation of these methods.
"publickey"	Public key authentication via the internal SSH server.
Other	Authentication with an unknown or unsupported method with this name was attempted via the internal SSH server.

If `success` is non-zero, the AAA authentication succeeded, and `groups` is an array of length `ngroups` that gives the groups that will be assigned to the user at login. If the callback returns CONFD_OK, the complete authentication succeeds and the user is logged in. If it returns CONFD_ERR (or an invalid return value), the authentication fails.

If `success` is zero, the AAA authentication failed, with the reason given by `logno` (one of `CONFD_BAD_LOCAL_PASS`, `CONFD_NO_SUCH_LOCAL_USER`, or `CONFD_SSH_NO_LOGIN`) and

the explanatory string `reason`. This invocation is only for informational purposes - the callback return value has no effect on the authentication, and should normally be CONFD_OK.

`void` **`confd_auth_seterr`**`(struct confd_auth_ctx *`*`actx`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide a text message when the callback returns CONFD_ERR. If used when rejecting a successful authentication, the message will be logged in ConfD's audit log (otherwise a generic "rejected by application callback" message is logged).

AUTHORIZATION CALLBACKS

We can register two authorization callbacks with ConfD's AAA subsystem. These will be invoked when the northbound agents check that a command or a data access is allowed by the AAA access rules. The callbacks can partially or completely replace the access checks done within the AAA subsystem, and they may accept or reject the access. Typically many access checks are done during the processing of commands etc, and using these callbacks can thus have a significant performance impact. Unless it is a requirement to query an external authorization mechanism, it is far better to only configure access rules in the AAA data model (see the AAA chapter in the User Guide).

The callbacks will only be invoked if they are both enabled via `/confdConfig/aaa/authorization/callback/enabled` in `confd.conf` (see confd.conf(5)) and registered as described here.

Note

If the callbacks are enabled in `confd.conf` but no registration has been done, or if invocation keeps failing for some reason, *all* access checks will be rejected.

`int` **`confd_register_authorization_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_authorization_cbs *`*`acb`*`);`

Registers the authorization callbacks. The struct confd_authorization_cbs is defined as:

```
struct confd_authorization_cbs {
    int cmd_filter;
    int data_filter;
    int (*chk_cmd_access)(struct confd_authorization_ctx *actx,
                          char **cmdtokens, int ntokens, int cmdop);
    int (*chk_data_access)(struct confd_authorization_ctx *actx,
                           u_int32_t hashed_ns, confd_hkeypath_t *hkp,
                           int dataop, int how);
};
```

Both callbacks are optional, i.e. we can set the function pointer in struct confd_authorization_cbs to NULL if we don't want the corresponding callback invocation. In this case the AAA subsystem will handle the access check as if the callback was registered, but always replied with `CONFD_ACCESS_RESULT_DEFAULT` (see below).

The `cmd_filter` and `data_filter` elements can be used to prevent access checks from causing invocation of a callback even though it is registered. If we do not want any filtering, they must be set to zero. The value is a bitmask obtained by ORing together values: For `cmd_filter`, we can use the possible values for *`cmdop`* (see below), preventing the corresponding invocations of `chk_cmd_access()`. For `data_filter`, we can use the possible values for *`dataop`* and *`how`* (see below), preventing the corresponding invocation of `chk_data_access()`. If the callback invocation is prevented by filtering, the AAA subsystem will handle the access check as if the callback had replied with `CONFD_ACCESS_RESULT_CONTINUE` (see below).

Both callbacks are invoked with a pointer to an authorization context that provides information about the user session that the access check pertains to, and the group list for that session. The struct confd_authorization_ctx is defined as:

```
struct confd_authorization_ctx {
    struct confd_user_info *uinfo;
    int ngroups;
    char **groups;
    struct confd_daemon_ctx *dx;
    /* ConfD internal fields */
    int result;
    int query_ref;
};
```

`chk_cmd_access()`
: This callback is invoked for command authorization, i.e. it corresponds to the rules under `/aaa/authorization/cmdrules` in the AAA data model. *`cmdtokens`* is an array of *`ntokens`* NUL-terminated strings representing the command to be checked, corresponding to the `command` leaf in the `cmdrule` list. If `/confdConfig/cli/modeInfoInAAA` is enabled in `confd.conf` (see confd.conf(5)), mode names will be prepended in the *`cmdtokens`* array. The *`cmdop`* parameter gives the operation, corresponding to the `ops` leaf in the `cmdrule` list. The possible values for *`cmdop`* are:

  `CONFD_ACCESS_OP_READ`
  : Read access. The CLI will use this during command completion, to filter out alternatives that are disallowed by AAA.

  `CONFD_ACCESS_OP_EXECUTE`
  : Execute access. This is used when a command is about to be executed.

  ## Note

  This callback may be invoked with `actx->uinfo == NULL`, meaning that no user session has been established for the user yet. This will occur e.g. when the CLI checks whether a user attempting to log in is allowed to (implicitly) execute the command "request system logout user" (J-CLI) or "logout" (C/I-CLI) when the maximum number of sessions has already been reached (if allowed, the CLI will ask whether the user wants to terminate one of the existing sessions).

`chk_data_access()`
: This callback is invoked for data authorization, i.e. it corresponds to the rules under `/aaa/authorization/datarules` in the AAA data model. *`hashed_ns`* and *`hkp`* give the namespace and hkeypath of the data node to to be checked, corresponding to the `namespace` and `keypath` leafs in the `datarule` list. The *`hkp`* parameter may be NULL, which means that access to the entire namespace given by *`hashed_ns`* is requested. When a hkeypath is provided, some key elements in the path may be without key values (i.e. hkp->v[n][0].type == C_NOEXISTS). This indicates "wildcard" keys, used for CLI tab completion when keys are not fully specified. The *`dataop`* parameter gives the operation, corresponding the `ops` leaf in the `datarule` list. The possible values for *`dataop`* are:

  `CONFD_ACCESS_OP_READ`
  : Read access.

  `CONFD_ACCESS_OP_EXECUTE`
  : Execute access.

  `CONFD_ACCESS_OP_CREATE`
  : Create access.

CONFD_ACCESS_OP_UPDATE
Update access.

CONFD_ACCESS_OP_DELETE
Delete access.

CONFD_ACCESS_OP_WRITE
Write access. This is used when the specific write operation (create/update/delete) isn't known yet, e.g. in CLI command completion or processing of a NETCONF **edit-config**.

The *how* parameter is one of:

CONFD_ACCESS_CHK_INTERMEDIATE
Access to the given data node *or* its descendants is requested. This is used e.g. in CLI command completion or processing of a NETCONF **edit-config**.

CONFD_ACCESS_CHK_FINAL
Access to the specific data node is requested.

int **confd_access_reply_result**(struct confd_authorization_ctx *actx, int result);

The callbacks must call this function to report the result of the access check to ConfD, and should normally return CONFD_OK. If any other value is returned, it will cause the access check to be rejected. The *actx* parameter is the pointer to the authorization context passed in the callback invocation, and *result* must be one of:

CONFD_ACCESS_RESULT_ACCEPT
The access is allowed. This is a "final verdict", analogous to a "full match" when the AAA rules are used.

CONFD_ACCESS_RESULT_REJECT
The access is denied.

CONFD_ACCESS_RESULT_CONTINUE
The access is allowed "so far". I.e. access to sub-elements is not necessarily allowed. This result is mainly useful when chk_cmd_access() is called with *cmdop* == CONFD_ACCESS_OP_READ or chk_data_access() is called with *how* == CONFD_ACCESS_CHK_INTERMEDIATE.

CONFD_ACCESS_RESULT_DEFAULT
The request should be handled according to the rules configured in the AAA data model.

int **confd_authorization_set_timeout**(struct confd_authorization_ctx *actx, int timeout_secs);

The authorization callbacks are invoked on the daemon control socket, and as such are expected to complete quickly, within the timeout specified for /confdConfig/capi/newSessionTimeout. However in case they send requests to a remote server, and such a request needs to be retried, this function can be used to extend the timeout for the current callback invocation. The timeout is given in seconds from the point in time when the function is called.

ERROR FORMATTING CALLBACK

It is possible to register a callback function to generate customized error messages for ConfD's internally generated errors. All the customizable errors are defined with a type and a code in the XML document $CONFD_DIR/src/confd/errors/errcode.xml in the ConfD release. To use this functionality,

the application must `#include` the file `confd_errcode.h`, which defines C constants for the types and codes.

`int` **`confd_register_error_cb`**`(struct confd_daemon_ctx *`*`dx`*`, const struct confd_error_cb *`*`ecb`*`);`

Registers the error formatting callback. The struct confd_error_cb is defined as:

```
struct confd_error_cb {
    int error_types;
    void (*format_error)(struct confd_user_info *uinfo,
                         struct confd_errinfo *errinfo,
                         char *default_msg);
};
```

The `error_types` element is the logical OR of the error types that the callback should handle. An application daemon can only register one error formatting callback, and only one daemon can register for each error type. The available types are:

`CONFD_ERRTYPE_VALIDATION`
: Errors detected by ConfD's internal semantic validation of the data model constraints, e.g. mandatory elements that are unset, dangling references, etc. The codes for this type are the `confd_errno` values corresponding to the validation errors, as resulting e.g. from a call to `maapi_apply_trans()` (see confd_lib_maapi(3)). I.e. CONFD_ERR_NOTSET, CONFD_ERR_BAD_KEYREF, etc - see the 'id' attribute in `errcode.xml`.

`CONFD_ERRTYPE_BAD_VALUE`
: Type errors, i.e. errors generated when an invalid value is given for a leaf in the data model. The codes for this type are defined in `confd_errcode.h` as CONFD_BAD_VALUE_XXX, where "XXX" is the all-uppercase form of the code name given in `errcode.xml`.

`CONFD_ERRTYPE_CLI`
: CLI-specific errors. The codes for this type are defined in `confd_errcode.h` as CONFD_CLI_XXX in the same way as for `CONFD_ERRTYPE_BAD_VALUE`.

`CONFD_ERRTYPE_MISC`
: Miscellaneous errors, which do not fit into the other categories. The codes for this type are defined in `confd_errcode.h` as CONFD_MISC_XXX in the same way as for `CONFD_ERRTYPE_BAD_VALUE`.

The `format_error()` callback is invoked with a pointer to a struct confd_errinfo, which gives the error type and type-specific structured information about the details of the error. It is defined as:

```
struct confd_errinfo {
    int type;  /* CONFD_ERRTYPE_XXX */
    union {
        struct confd_errinfo_validation validation;
        struct confd_errinfo_bad_value bad_value;
        struct confd_errinfo_cli cli;
        struct confd_errinfo_misc misc;
    } info;
};
```

For `CONFD_ERRTYPE_VALIDATION`, the `struct confd_errinfo_validation validation` gives the detailed information, using an `info` union that has a specific struct member for each code:

```
struct confd_errinfo_validation {
    int code;  /* CONFD_ERR_NOTSET, CONFD_ERR_TOO_FEW_ELEMS, ... */
```

```
    union {
        struct {
            /* the element given by kp is not set */
            confd_hkeypath_t *kp;
        } notset;
        struct {
            /* kp has n instances, must be at least min */
            confd_hkeypath_t *kp;
            int n, min;
        } too_few_elems;
        struct {
            /* kp has n instances, must be at most max */
            confd_hkeypath_t *kp;
            int n, max;
        } too_many_elems;
        struct {
            /* the elements given by kps1 have the same set
               of values vals as the elements given by kps2
               (kps1, kps2, and vals point to n_elems long arrays) */
            int n_elems;
            confd_hkeypath_t *kps1;
            confd_hkeypath_t *kps2;
            confd_value_t *vals;
        } non_unique;
        struct {
            /* the element given by kp references
               the non-existing element given by ref
               Note: 'ref' may be NULL or have key elements without values
               (ref->v[n][0].type == C_NOEXISTS) if it cannot be instantiated */
            confd_hkeypath_t *kp;
            confd_hkeypath_t *ref;
        } bad_keyref;
        struct {
            /* the mandatory 'choice' statement choice in the
               container kp does not have a selected 'case' */
            confd_value_t *choice;
            confd_hkeypath_t *kp;
        } unset_choice;
        struct {
            /* the 'must' expression expr for element kp is not satisfied
               - error_message and and error_app_tag are NULL if not given
               in the 'must'; val points to the value of the element if it
               has one, otherwise it is NULL */
            char *expr;
            confd_hkeypath_t *kp;
            char *error_message;
            char *error_app_tag;
            confd_value_t *val;
        } must_failed;
        struct {
            /* the element kp has the instance-identifier value instance,
               which doesn't exist, but require-instance is 'true' */
            confd_hkeypath_t *kp;
            confd_hkeypath_t *instance;
        } missing_instance;
        struct {
            /* the element kp has the instance-identifier value instance,
               which doesn't conform to the specified path filters */
            confd_hkeypath_t *kp;
            confd_hkeypath_t *instance;
```

```
        } invalid_instance;
        struct {
            /* the expression for a configuration policy rule evaluated to
               'false' - error_message is the associated error message */
            char *error_message;
        } policy_failed;
        struct {
            /* the XPath expression expr, for the configuration policy
               rule with key name, could not be compiled due to msg */
            char *name;
            char *expr;
            char *msg;
        } policy_compilation_failed;
        struct {
            /* the expression expr, for the configuration policy rule
               with key name, failed XPath evaluation due to msg */
            char *name;
            char *expr;
            char *msg;
        } policy_evaluation_failed;
    } info;
    int test;            /* 1 if 'validate', 0 if 'commit' */
    struct confd_trans_ctx *tctx; /* only valid for duration of callback */
};
```

The member structs are named as the `confd_errno` values that are used for the `code` elements, i.e. `notset` for CONFD_ERR_NOTSET, etc. For this error type, the callback also has full information about the transaction that failed validation via the `struct confd_trans_ctx *tctx` element - it is even possible to use `maapi_attach()` (see confd_lib_maapi(3)) to attach to the transaction and read arbitrary data from it, in case the data directly related to the error (as given in the code-specific struct) is not sufficient.

For the other error types, the corresponding `confd_errinfo_xxx` struct gives the code and an array with the parameters for the default error message, as defined by the `<fmt>` element in `errcode.xml`:

```
enum confd_errinfo_ptype {
    CONFD_ERRINFO_KEYPATH,
    CONFD_ERRINFO_STRING
};
```

```
struct confd_errinfo_param {
    enum confd_errinfo_ptype type;
    union {
        confd_hkeypath_t *kp;
        char *str;
    } val;
};
```

```
struct confd_errinfo_bad_value {
    int code;
    int n_params;
    struct confd_errinfo_param *params;
};
```

The parameters in the `params` array are given in the order they appear in the `<fmt>` specification. Parameters that are specified as `{path}` have `params[n].type` set to `CONFD_ERRINFO_KEYPATH`, and are represented as a confd_hkeypath_t that can be accessed via `params[n].val.kp`. All other parameters are represented as strings, i.e. `params[n].type` is `CONFD_ERRINFO_STR` and the string value can be accessed via `params[n].val.str`. The `struct confd_errinfo_cli`

`cli` and `struct confd_errinfo_misc misc` union members have the same form as `struct confd_errinfo_bad_value` shown above.

Finally, the *`default_msg`* callback parameter gives the default error message that will be reported to the user if the `format_error()` function does not generate a replacement.

`void` **`confd_error_seterr`**`(struct confd_user_info *`*`uinfo`*`, const char *`*`fmt`*`, ...);`

This function must be called by `format_error()` to provide a replacement of the default error message. If `format_error()` returns without calling `confd_error_seterr()`, the default message will be used.

Here is an example that targets a specific validation error for a specific element in the data model. For this case only, it replaces ConfD's internally generated messages of the form:

```
"too many 'protocol bgp', 2 configured, at most 1 must be configured"
```

with

```
"Only 1 bgp instance is supported, cannot define 2"
```

```
#include <confd_lib.h>
#include <confd_dp.h>
#include <confd_errcode.h>
.
.
int main(int argc, char **argv)
{
     struct confd_error_cb ecb;
     .
     .
     memset(&ecb, 0, sizeof(ecb));
     ecb.error_types = CONFD_ERRTYPE_VALIDATION;
     ecb.format_error = format_error;
     if (confd_register_error_cb(dctx, &ecb) != CONFD_OK)
          confd_fatal("Couldn't register error callback\n");
     .
}

static void format_error(struct confd_user_info *uinfo,
                         struct confd_errinfo *errinfo,
                         char *default_msg)
{
     struct confd_errinfo_validation *err;
     confd_hkeypath_t *kp;

     err = &errinfo->info.validation;
     if (err->code == CONFD_ERR_TOO_MANY_ELEMS) {
          kp = err->info.too_many_elems.kp;
          if (CONFD_GET_XMLTAG(&kp->v[0][0]) == myns_bgp &&
              CONFD_GET_XMLTAG(&kp->v[1][0]) == myns_protocol) {
              confd_error_seterr(uinfo,
                                 "Only %d bgp instance is supported, "
                                 "cannot define %d",
                                 err->info.too_many_elems.max,
                                 err->info.too_many_elems.n);
          }
     }
```

```
}
```

The CLI-specific "Aborted: " prefix is not included in the message for this error type - if we wanted to replace that too, we could include the `CONFD_ERRTYPE_CLI` error type in the registration and process the `CONFD_CLI_COMMAND_ABORTED` error code for this type, see `errcode.xml`.

SEE ALSO

confd.conf(5) - ConfD daemon configuration file format

The ConfD User Guide

Name

confd_lib_events — library for subscribing to ConfD event notifications

Synopsis

#include <confd_lib.h> #include <confd_events.h>

```
int confd_notifications_connect(int sock, const struct sockaddr* srv,
int srv_sz, int mask);

int confd_notifications_connect2(int sock, const struct sockaddr* srv,
int srv_sz, int mask, struct confd_notifications_data *data);

int confd_read_notification(int sock, struct confd_notification *n);

void confd_free_notification(struct confd_notification *n);

int confd_diff_notification_done(int sock, struct confd_trans_ctx *tc-
tx);

int confd_sync_audit_notification(int sock, int usid);

int confd_sync_ha_notification(int sock);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to ConfD and subscribe to certain events generated by ConfD. The API to receive events from ConfD is a socket based API whereby the application connects to ConfD and receives events on a socket. See also the Notifications chapter in the User Guide. The program `misc/notifications/confd_notifications.c` in the examples collection illustrates subscription and processing for all these events, and can also be used standalone in a development environment to monitor ConfD events.

EVENTS

The following events can be subscribed to:

`CONFD_NOTIF_AUDIT`
: All audit log events are sent from ConfD on the event notification socket.

`CONFD_NOTIF_AUDIT_SYNC`
: This flag modifies the behavior of a subscription for the `CONFD_NOTIF_AUDIT` event - it has no effect unless `CONFD_NOTIF_AUDIT` is also present. If this flag is present, ConfD will stop processing in the user session that causes an audit notification to be sent, and continue processing in that user session only after all subscribers with this flag have called `confd_sync_audit_notification()`.

`CONFD_NOTIF_DAEMON`
: All log events that also goes to the `/confdConf/logs/confdLog` log are sent from ConfD on the event notification socket.

CONFD_NOTIF_NETCONF
All log events that also goes to the /confdConf/logs/netconfLog log are sent from ConfD on the event notification socket.

CONFD_NOTIF_DEVEL
All log events that also goes to the /confdConf/logs/developerLog log are sent from ConfD on the event notification socket.

CONFD_NOTIF_TAKEOVER_SYSLOG
If this flag is present, ConfD will stop syslogging. The idea behind the flag is that we want to configure syslogging for ConfD in order to let ConfD log its startup sequence. Once ConfD is started we wish to subsume the syslogging done by ConfD. Typical applications that use this flag want to pick up all log messages, reformat them and use some local logging method.

Once all subscriber sockets with this flag set are closed, ConfD will resume to syslog.

CONFD_NOTIF_COMMIT_SIMPLE
An event indicating that a user has somehow modified the configuration.

CONFD_NOTIF_COMMIT_DIFF
An event indicating that a user has somehow modified the configuration. The main difference between this event and the abovementioned CONFD_NOTIF_COMMIT_SIMPLE is that this event is synchronous, i.e. the entire transaction hangs until we have explicitly called confd_diff_notification_done(). The purpose of this event is to give the applications a chance to read the configuration diffs from the transaction before it finishes. A user subscribing to this event can use MAAPI to attach (maapi_attach()) to the running transaction and use maapi_diff_iterate() to iterate through the diff. This feature can also be used to produce a complete audit trail of who changed what and when in the system. It is up to the application to format that audit trail.

CONFD_NOTIF_COMMIT_FAILED
This event is generated when a data provider fails in its commit callback. ConfD executes a two-phase commit procedure towards all data providers when committing transactions. When a provider fails in commit, the system is an unknown state. See confd_lib_maapi(3) and the function maapi_get_running_db_state(). If the provider is "external", the name of failing daemon is provided. If the provider is another NETCONF agent, the IP address and port of that agent is provided.

CONFD_NOTIF_CONFIRMED_COMMIT
This event is generated when a user has started a confirmed commit, when a confirming commit is issued, or when a confirmed commit is aborted; represented by enum confd_confirmed_commit_type.

For a confirmed commit, the timeout value is also present in the notification.

CONFD_NOTIF_COMMIT_PROGRESS
This event provides progress information about the commit of a transaction, i.e. the same information that is reported when the **commit | details** CLI command is used. The application receives a struct confd_progress_notification which gives details for the specific transaction along with the progress information, see confd_events.h.

CONFD_NOTIF_USER_SESSION
An event related to user sessions. There are 6 different user session related event types, defined in enum confd_user_sess_type: session starts/stops, session locks/unlocks database, session starts/stop database transaction.

CONFD_NOTIF_HA_INFO
An event related to ConfDs perception of the current cluster configuration.

CONFD_NOTIF_HA_INFO_SYNC
: This flag modifies the behavior of a subscription for the CONFD_NOTIF_HA_INFO event - it has no effect unless CONFD_NOTIF_HA_INFO is also present. If this flag is present, ConfD will stop all HA processing, and continue only after all subscribers with this flag have called confd_sync_ha_notification().

CONFD_NOTIF_SUBAGENT_INFO
: Only sent if ConfD runs as a master agent with subagents enabled. This event is sent when the subagent connection is lost or reestablished. There are two event types, defined in enum confd_subagent_info_type: subagent up and subagent down.

CONFD_NOTIF_SNMPA
: This event is generated whenever an SNMP pdu is processed by ConfD. The application receives a struct confd_snmpa_notification structure. The structure contains a series of fields describing the sent or received SNMP pdu. It contains a list of all varbinds in the pdu.

  Each varbind contains a confd_value_t with the string representation of the SNMP value. Thus the type of the value in a varbind is always C_BUF. See confd_events.h include file for the details of the received structure.

  ## Note

  This event may allocate memory dynamically inside the struct confd_notification, thus we must always call confd_free_notification() after receiving and processing this event.

CONFD_NOTIF_FORWARD_INFO
: This event is generated whenever whenever ConfD forwards (proxies) a northbound agent.

CONFD_NOTIF_UPGRADE_EVENT
: This event is generated for the different phases of an in-service upgrade, i.e. when the data model is upgraded while ConfD is running. The application receives a struct confd_upgrade_notification where the enum confd_upgrade_event_type event gives the specific upgrade event, see confd_events.h. The events correspond to the invocation of the MAAPI functions that drive the upgrade, see confd_lib_maapi(3).

CONFD_NOTIF_HEARTBEAT
: This event can be be used by applications that wish to monitor the health and liveness of ConfD itself. It needs to be requested through a call to confd_notifications_connect2(), where the required heartbeat_interval can be provided via the *struct confd_notifications_data* parameter. ConfD will continuously generate heartbeat events on the notification socket. If ConfD fails to do so, ConfD is hung, or prevented from getting the CPU time required to send the event. The timeout interval is measured in milliseconds. Recommended value is 10000 milliseconds to cater for truly high load situations. Values less than 1000 are changed to 1000.

CONFD_NOTIF_HEALTH_CHECK
: This event is similar to CONFD_NOTIF_HEARTBEAT, in that it can be be used by applications that wish to monitor the health and liveness of ConfD itself. However while CONFD_NOTIF_HEARTBEAT will be generated as long as ConfD is not completely hung, CONFD_NOTIF_HEALTH_CHECK will only be generated after a basic liveness check of the different ConfD subsystems has completed successfully. This event also needs to be requested through a call to confd_notifications_connect2(), where the required health_check_interval can be provided via the *struct confd_notifications_data* parameter. Since the event generation incurs more processing than CONFD_NOTIF_HEARTBEAT, a longer interval than 10000 milliseconds is recommended, but in particular the application must be prepared for the actual interval

to be significantly longer than the requested one in high load situations. Values less than 1000 are changed to 1000.

`NCS_NOTIF_PACKAGE_RELOAD`
This event is generated whenever whenever NCS has completed a package reload.

`NCS_NOTIF_CQ_PROGRESS`
This event is generated to report the progress of commit queue entries.

The application receives a struct ncs_cq_progress_notification where the enum ncs_cq_progress_notif_type type gives the specific event that occurred, see `confd_events.h`. This can be one of `NCS_CQ_ITEM_WAITING` - (waiting on another executing entry), `NCS_CQ_ITEM_EXECUTING`, `NCS_CQ_ITEM_LOCKED` (stalled by parent queue in cluster), `NCS_CQ_ITEM_COMPLETED`, `NCS_CQ_ITEM_FAILED` or `NCS_CQ_ITEM_DELETED`.

`CONFD_NOTIF_STREAM_EVENT`
This event is generated for a notification stream, i.e. event notifications sent by an application as described in the NOTIFICATION STREAMS section of confd_lib_dp(3). The application receives a struct confd_stream_notification where the enum confd_stream_notif_type type gives the specific event that occurred, see `confd_events.h`. This can be either an actual event notification (`CONFD_STREAM_NOTIFICATION_EVENT`), one of `CONFD_STREAM_NOTIFICATION_COMPLETE` or `CONFD_STREAM_REPLAY_COMPLETE`, which indicates that a requested replay has completed, or `CONFD_STREAM_REPLAY_FAILED`, which indicates that a requested replay could not be carried out. In all cases except `CONFD_STREAM_NOTIFICATION_EVENT`, no further `CONFD_NOTIF_STREAM_EVENT` events will be delivered on the socket.

This event also needs to be requested through a call to `confd_notifications_connect2()`, where the required `stream_name` must be provided via the *`struct confd_notifications_data`* parameter. The additional elements in the struct can be used as follows:

- The `start_time` element can be given to request a replay, in which case `stop_time` can also be given to specify the end of the replay (or "live feed"). The `start_time` and `stop_time` must be set to the type C_NOEXISTS to indicate that no value is given, otherwise values of type C_DATETIME must be given.

- The `xpath_filter` element may be used to specify an XPath filter to be applied to the notification stream. If no filtering is wanted, `xpath_filter` must be set to NULL.

- The `usid` element may be used to specify the id of an existing user session for filtering based on AAA rules. Only notifications that are allowed by the access rights of that user session will be received. If no AAA restrictions are wanted, `usid` must be set to `0`.

Note

This event may allocate memory dynamically inside the struct confd_notification, thus we must always call `confd_free_notification()` after receiving and processing this event.

Several of the above notification messages contain a lognumber which identifies the event. All log numbers are listed in the file `confd_logsyms.h`. Furthermore the array `confd_log_symbols[]` can be indexed with the lognumber and it contains the symbolic name of each error. The array `confd_log_descriptions[]` can also be indexed with the lognumber and it contains a textual description of the logged event.

FUNCTIONS

The API to receive events from ConfD is:

```
int confd_notifications_connect(int sock, const struct sockaddr* srv,
int srv_sz, int mask);

int confd_notifications_connect2(int sock, const struct sockaddr* srv,
int srv_sz, int mask, struct confd_notifications_data *data);
```

These functions create a notification socket. The *mask* is a bitmask of one or several enum confd_notification_type values:

```
enum confd_notification_type {
    CONFD_NOTIF_AUDIT                 = (1 << 0),
    CONFD_NOTIF_DAEMON                = (1 << 1),
    CONFD_NOTIF_TAKEOVER_SYSLOG       = (1 << 2),
    CONFD_NOTIF_COMMIT_SIMPLE         = (1 << 3),
    CONFD_NOTIF_COMMIT_DIFF           = (1 << 4),
    CONFD_NOTIF_USER_SESSION          = (1 << 5),
    CONFD_NOTIF_HA_INFO               = (1 << 6),
    CONFD_NOTIF_SUBAGENT_INFO         = (1 << 7),
    CONFD_NOTIF_COMMIT_FAILED         = (1 << 8),
    CONFD_NOTIF_SNMPA                 = (1 << 9),
    CONFD_NOTIF_FORWARD_INFO          = (1 << 10),
    CONFD_NOTIF_NETCONF               = (1 << 11),
    CONFD_NOTIF_DEVEL                 = (1 << 12),
    CONFD_NOTIF_HEARTBEAT             = (1 << 13),
    CONFD_NOTIF_CONFIRMED_COMMIT      = (1 << 14),
    CONFD_NOTIF_UPGRADE_EVENT         = (1 << 15),
    CONFD_NOTIF_COMMIT_PROGRESS       = (1 << 16),
    CONFD_NOTIF_AUDIT_SYNC            = (1 << 17),
    CONFD_NOTIF_HEALTH_CHECK          = (1 << 18),
    CONFD_NOTIF_STREAM_EVENT          = (1 << 19),
    CONFD_NOTIF_HA_INFO_SYNC          = (1 << 20),
    NCS_NOTIF_PACKAGE_RELOAD          = (1 << 21),
    NCS_NOTIF_CQ_PROGRESS             = (1 << 22)
};
```

The `confd_notifications_connect2()` variant is required if we wish to subscribe to `CONFD_NOTIF_HEARTBEAT`, `CONFD_NOTIF_HEALTH_CHECK`, or `CONFD_NOTIF_STREAM_EVENT` events. The struct confd_notifications_data is defined as:

```
struct confd_notifications_data {
    int heartbeat_interval;     /* required if we wish to generate */
                                /* CONFD_NOTIF_HEARTBEAT events    */
                                /* the time is milli seconds       */
    int health_check_interval;  /* required if we wish to generate */
                                /* CONFD_NOTIF_HEALTH_CHECK events */
                                /* the time is milli seconds       */
    /* The following five are used for CONFD_NOTIF_STREAM_EVENT    */
    char *stream_name;          /* stream name (required)          */
    confd_value_t start_time;   /* type = C_NOEXISTS or C_DATETIME */
    confd_value_t stop_time;    /* type = C_NOEXISTS or C_DATETIME */
                                /* when start_time is C_DATETIME   */
    char *xpath_filter;         /* optional XPath filter for the   */
                                /* stream -  NULL for no filter    */
    int usid;                   /* optional user session id for    */
```

```
                                /* AAA  restriction - 0 for no AAA */
};
```

When requesting the `CONFD_NOTIF_STREAM_EVENT` event, `confd_notifications_connect2()` may fail and return CONFD_ERR, with some specific `confd_errno` values:

`CONFD_ERR_NOEXISTS` The stream name given by `stream_name` does not exist.

`CONFD_ERR_XPATH` The XPath filter provided via `xpath_filter` failed to compile.

`CONFD_ERR_NOSESSION` The user session id given by `usid` does not identify an existing user session.

Note

If these calls fail (i.e. do not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

`int` **`confd_read_notification`**`(int` *`sock`*`, struct confd_notification *`*`n`*`);`

The application is responsible for polling the notification socket. Once data is available to be read on the socket the application must call `confd_read_notification()` to read the data from the socket. On success the function returns CONFD_OK and populates the struct confd_notification* pointer. See `confd_events.h` for the definition of the struct confd_notification structure.

If the application is not reading from the socket and a write() from ConfD hangs for more than 15 seconds, ConfD will close the socket and log the event to the confdLog

`void` **`confd_free_notification`**`(struct confd_notification *`*`n`*`);`

The struct confd_notification can sometimes have memory dynamically allocated inside it. Currently the notification types that render structures with allocated memory inside them are `CONFD_NOTIF_SNMPA` and `CONFD_NOTIF_STREAM_EVENT`. If such an event is received, this function must be called to free any memory allocated inside the received notification structure.

For those notification structures that do not have any memory allocated, this function is a no-op, thus it is always safe to call this function after a notification structure has been processed.

`int` **`confd_diff_notification_done`**`(int` *`sock`*`, struct confd_trans_ctx *`*`tc-tx`*`);`

If the received event was CONFD_NOTIF_COMMIT_DIFF it is important that we call this function when we are done reading the transaction diffs over MAAPI. The transaction is hanging until this function gets called. This function also releases memory associated to the transaction in the library.

`int` **`confd_sync_audit_notification`**`(int` *`sock`*`, int` *`usid`*`);`

If the received event was CONFD_NOTIF_AUDIT, and we are subscribing to notifications with the flag CONFD_NOTIF_AUDIT_SYNC, this function must be called when we are done processing the notification. The user session is hanging until this function gets called.

`int` **`confd_sync_ha_notification`**`(int` *`sock`*`);`

If the received event was CONFD_NOTIF_HA_INFO, and we are subscribing to notifications with the flag CONFD_NOTIF_HA_INFO_SYNC, this function must be called when we are done processing the notification. All HA processing is blocked until this function gets called.

SEE ALSO

The ConfD User Guide

Name

confd_lib_ha — library for connecting to ConfD HA subsystem

Synopsis

#include <confd_lib.h> #include <confd_ha.h>

```
int confd_ha_connect(int sock, const struct sockaddr* srv, int srv_sz,
const char *token);

int confd_ha_bemaster(int sock, confd_value_t *mynodeid);

int confd_ha_beslave(int sock, confd_value_t *mynodeid, struct
confd_ha_node *master, int waitreply);

int confd_ha_berelay(int sock);

int confd_ha_benone(int sock);

int confd_ha_status(int sock, struct confd_ha_status *stat);

int confd_ha_slave_dead(int sock, confd_value_t *nodeid);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to the ConfD High Availability (HA) subsystem. ConfD can replicate the configuration data on several nodes in a cluster. The purpose of this API is to manage the HA functionality. The details on usage of the HA API are described in the chapter High availability in the User Guide.

FUNCTIONS

```
int confd_ha_connect(int sock, const struct sockaddr* srv, int srv_sz,
const char *token);
```

Connect a HA socket which can be used to control a ConfD HA node. The token is a secret string that must be shared by all participants in the cluster. There can only be one HA socket towards ConfD, a new call to `confd_ha_connect()` makes ConfD close the previous connection and reset the token to the new value. Returns CONFD_OK or CONFD_ERR.

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

```
int confd_ha_bemaster(int sock, confd_value_t *mynodeid);
```

Instruct a HA node to be master and also give the node a name. Returns CONFD_OK or CONFD_ERR.

Errors: CONFD_ERR_HA_BIND if we cannot bind the TCP socket, CONFD_ERR_BADSTATE if ConfD is still in start phase 0.

`int` **`confd_ha_beslave`**`(int` *`sock`*`, confd_value_t *`*`mynodeid`*`, struct confd_ha_node *`*`master`*`, int` *`waitreply`*`);`

Instruct a ConfD HA node to be slave with a named master. The *`waitreply`* is a boolean int. If 1, the function is synchronous and it will hang until the node has initialized its CDB database. This may mean that the CDB database is copied in its entirety from the master. If 0, we do not wait for the reply, but it is possible to use a notifications socket and get notified asynchronously via a HA_INFO_BESLAVE_RESULT notification. In both cases, it is also possible to use a notifications socket and get notified asynchronously when CDB at the slave is initialized.

If the call of this function fails with `confd_errno` CONFD_ERR_HA_CLOSED, it means that the initial synchronization with the master failed, either due to the socket being closed or due to a timeout while waiting for a response from the master. The function will fail with error CONFD_ERR_BADSTATE if ConfD is still in start phase 0.

Errors: CONFD_ERR_HA_CONNECT, CONFD_ERR_HA_BADNAME, CONFD_ERR_HA_BADTOKEN, CONFD_ERR_HA_BADFXS, CONFD_ERR_HA_BADVSN, CONFD_ERR_HA_CLOSED, CONFD_ERR_BADSTATE

`int` **`confd_ha_berelay`**`(int` *`sock`*`);`

Instruct an established HA slave node to be a relay for other slaves. This can be useful in certain deployment scenarios, but makes the management of the cluster more complex. Read more about this in the Relay slaves section of the High availability chapter in the User Guide. Returns CONFD_OK or CONFD_ERR.

Errors: CONFD_ERR_HA_BIND if we cannot bind the TCP socket, CONFD_ERR_BADSTATE if the node is not already a slave.

`int` **`confd_ha_benone`**`(int` *`sock`*`);`

Instruct a node to resume the initial state, i.e. neither master nor slave.

Errors: CONFD_ERR_BADSTATE if ConfD is still in start phase 0.

`int` **`confd_ha_status`**`(int` *`sock`*`, struct confd_ha_status *`*`stat`*`);`

Query a ConfD HA node for its status. If successful, the function populates the confd_ha_status structure. This is the only HA related function which is possible to call while the ConfD daemon is still in start phase 0.

`int` **`confd_ha_slave_dead`**`(int` *`sock`*`, confd_value_t *`*`nodeid`*`);`

This function must be used by the application to inform ConfD HA subsystem that another node which is possibly connected to ConfD is dead.

Errors: CONFD_ERR_BADSTATE if ConfD is still in start phase 0.

SEE ALSO

confd.conf(5) - ConfD daemon configuration file format

The ConfD User Guide

Name

confd_lib_lib — common library functions for applications connecting to ConfD

Synopsis

```
#include <confd_lib.h>

void confd_init(const char *name, FILE *estream, const enum
confd_debug_level debug);

int confd_set_debug(enum confd_debug_level debug, FILE *estream);

void confd_fatal(const char *fmt, ...);

int confd_load_schemas(const struct sockaddr* srv, int srv_sz);

int confd_load_schemas_list(const struct sockaddr* srv, int srv_sz, int
flags, const u_int32_t *nshash, const int *nsflags, int num_ns);

int confd_mmap_schemas_setup(void *addr, size_t size, const char *file-
name, int flags);

int confd_mmap_schemas(const char *filename);

int confd_svcmp(const char *s, const confd_value_t *v);

int confd_pp_value(char *buf, int bufsiz, const confd_value_t *v);

int confd_ns_pp_value(char *buf, int bufsiz, const confd_value_t *v,
int ns);

int confd_pp_kpath(char *buf, int bufsiz, const confd_hkeypath_t *hkey-
path);

int confd_pp_kpath_len(char *buf, int bufsiz, const confd_hkeypath_t
*hkeypath, int len);

char *confd_xmltag2str(u_int32_t ns, u_int32_t xmltag);

int confd_xpath_pp_kpath(char *buf, int bufsiz, u_int32_t ns, const
confd_hkeypath_t *hkeypath);

int confd_format_keypath(char *buf, int bufsiz, const char *fmt, ...);

int confd_vformat_keypath(char *buf, int bufsiz, const char *fmt,
va_list ap);

int confd_get_nslist(struct confd_nsinfo **listp);

char *confd_ns2prefix(u_int32_t ns);

char *confd_hash2str(u_int32_t hash);

u_int32_t confd_str2hash(const char *str);

struct confd_cs_node *confd_find_cs_root(int ns);
```

struct confd_cs_node *__confd_find_cs_node__(const confd_hkeypath_t *_hkeypath_, int _len_);

struct confd_cs_node *__confd_find_cs_node_child__(const struct confd_cs_node *_parent_, struct xml_tag _xmltag_);

struct confd_cs_node *__confd_cs_node_cd__(const struct confd_cs_node *_start_, const char *_fmt_, ...);

int __confd_max_object_size__(struct confd_cs_node *_object_);

struct confd_cs_node *__confd_next_object_node__(struct confd_cs_node *_object_, struct confd_cs_node *_cur_, confd_value_t *_value_);

struct confd_type *__confd_find_ns_type__(u_int32_t _nshash_, const char *_name_);

struct confd_type *__confd_get_leaf_list_type__(struct confd_cs_node *_node_);

int __confd_val2str__(struct confd_type *_type_, const confd_value_t *_val_, char *_buf_, int _bufsiz_);

int __confd_str2val__(struct confd_type *_type_, const char *_str_, confd_value_t *_val_);

char *__confd_val2str_ptr__(struct confd_type *_type_, const confd_value_t *_val_);

int __confd_get_decimal64_fraction_digits__(struct confd_type *_type_);

int __confd_hkp_tagmatch__(struct xml_tag _tags[]_, int _tagslen_, confd_hkeypath_t *_hkp_);

int __confd_hkp_prefix_tagmatch__(struct xml_tag _tags[]_, int _tagslen_, confd_hkeypath_t *_hkp_);

int __confd_val_eq__(const confd_value_t *_v1_, const confd_value_t *_v2_);

void __confd_free_value__(confd_value_t *_v_);

confd_value_t *__confd_value_dup_to__(const confd_value_t *_v_, confd_value_t *_newv_);

void __confd_free_dup_to_value__(confd_value_t *_v_);

confd_value_t *__confd_value_dup__(const confd_value_t *_v_);

void __confd_free_dup_value__(confd_value_t *_v_);

confd_hkeypath_t *__confd_hkeypath_dup__(const confd_hkeypath_t *_src_);

confd_hkeypath_t *__confd_hkeypath_dup_len__(const confd_hkeypath_t *_src_, int _len_);

void __confd_free_hkeypath__(confd_hkeypath_t *_hkp_);

```
void confd_free_authorization_info(struct confd_authorization_info
*ainfo);

char *confd_lasterr(void);

char *confd_strerror(int code);

struct xml_tag *confd_last_error_apptag(void);

int confd_register_ns_type(u_int32_t nshash, const char *name, struct
confd_type *type);

int confd_register_node_type(struct confd_cs_node *node, struct
confd_type *type);

int confd_type_cb_init(struct confd_type_cbs **cbs);

int confd_decrypt(const char *ciphertext, int len, char *output);

int confd_stream_connect(int sock, const struct sockaddr* srv, int
srv_sz, int id, int flags);

int confd_deserialize(struct confd_deserializable *s, unsigned char
*buf);

int confd_serialize(struct confd_serializable *s, unsigned char *buf,
int bufsz, int *bytes_written, unsigned char **allocated);

void confd_deserialized_free(struct confd_deserializable *s);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to ConfD. This manual page describes functions and data structures that are not specific to any one of the APIs that are described in the other confd_lib_xxx(3) manual pages.

FUNCTIONS

```
void confd_init(const char *name, FILE *estream, const enum
confd_debug_level debug);
```

Initializes the ConfD library. Must be called before any other ConfD API functions are called.

The *debug* parameter is used to control the debug level. The following levels are available:

`CONFD_SILENT` No printouts whatsoever are produced by the library.

`CONFD_DEBUG` Various printouts will occur for various error conditions. This is a decent value to have as default. If syslog is enabled for the library, these printouts will be logged at syslog level `LOG_ERR`, except for errors where `confd_errno` is `CONFD_ERR_INTERNAL`, which are logged at syslog level `LOG_CRIT`.

CONFD_TRACE — The execution of callback functions and CDB/MAAPI API calls will be traced. This is very verbose and very useful during debugging. If syslog is enabled for the library, these printouts will be logged at syslog level LOG_DEBUG.

CONFD_PROTO_TRACE — The low-level protocol exchange between the application and ConfD will be traced. This is even more verbose than CONFD_TRACE, and normally only of interest to Tail-f support. These printouts will not be logged via syslog, i.e. a non-NULL value for the *estream* parameter must be provided.

The *estream* parameter is used by all printouts from the library. The *name* parameter is typically included in most of the debug printouts. If the *estream* parameter is NULL, no printouts to a file will occur. Independent of the *estream* parameter, syslog can be enabled for the library by setting the global variable confd_lib_use_syslog to 1. See SYSLOG AND DEBUG in this man page.

```
int confd_set_debug(enum confd_debug_level debug, FILE *estream);
```

This function can be used to change the *estream* and *debug* parameters for the library.

```
int confd_load_schemas(const struct sockaddr* srv, int srv_sz);
```

Utility function that uses maapi_load_schemas() (see confd_lib_maapi(3)) to load schema information from ConfD. This function connects to ConfD and loads all the schema information in ConfD for all loaded "fxs" files into the library. This is necessary in order to get proper printouts of e.g. confd_hkeypaths which otherwise just contains arrays of integers. This function should typically always be called when we initialize the library. See confd_types(3).

```
int confd_load_schemas_list(const struct sockaddr* srv, int srv_sz, int
flags, const u_int32_t *nshash, const int *nsflags, int num_ns);
```

Utility function that uses maapi_load_schemas_list() to load a subset of the schema information from ConfD. See the description of maapi_load_schemas_list() in confd_lib_maapi(3) for the details of how to use the *flags*, *nshash*, *nsflags*, and *num_ns* parameters.

```
int confd_mmap_schemas_setup(void *addr, size_t size, const char *file-
name, int flags);
```

This function sets up for a subsequent call of one of the schema-loading functions (confd_load_schemas() etc) to load the schema information into a shared memory segment instead of into the process' heap. See the section Using shared memory for schema information in the Advanced Topics chapter in the User Guide for usage discussion. The *addr* and (potentially) *size* arguments are passed to mmap(2), and *filename* specifies the pathname of a file to use as backing store. The *flags* parameter can be given as CONFD_MMAP_SCHEMAS_KEEP_SIZE to request that the shared memory segment should be exactly the size given by the (non-zero) *size* argument - if this size is insufficient to hold the schema information, the schema-loading function will fail.

```
int confd_mmap_schemas(const char *filename);
```

Map a shared memory segment, previously created by confd_mmap_schemas_setup() and subsequent schema loading, into the current process' address space, and make it ready for use. The *filename* argument specifies the pathname of the file that is used as backing store.

```
int confd_svcmp(const char *s, const confd_value_t *v);
```

Utility function with similar semantics to strcmp() which compares a confd_value_t to a char*.

```
int confd_pp_value(char *buf, int bufsiz, const confd_value_t *v);
```

Utility function which pretty prints up to *bufsiz* characters into *buf*, giving a string representation of the value *v*. Since only the "primitive" type as defined by the enum confd_vtype is available, confd_pp_value() can not produce a true string representation in all cases, see the list below. If this is a problem, use confd_val2str() instead.

C_ENUM_VALUE The value is printed as "enum<N>", where N is the integer value.

C_BIT32 The value is printed as "bits<X>", where X is an unsigned integer in hexadecimal format.

C_BIT64 The value is printed as "bits<X>", where X is an unsigned integer in hexadecimal format.

C_BINARY The string representation for xs:hexBinary is used, i.e. a sequence of hexadecimal characters.

C_DECIMAL64 If the value of the fraction_digits element is within the possible range (1..18), it is assumed to be correct for the type and used for the string representation. Otherwise the value is printed as "invalid64<N>", where N is the value of the value element.

C_XMLTAG The string representation is printed if schema information has been loaded into the library. Otherwise the value is printed as "tag<N>", where N is the integer value.

C_IDENTITYREF The string representation is printed if schema information has been loaded into the library. Otherwise the value is printed as "idref<N>", where N is the integer value.

All the pp pretty print functions, i.e. confd_pp_value() confd_ns_pp_value(), confd_pp_kpath() and confd_xpath_pp_kpath(), as well as the confd_format_keypath() and confd_val2str() functions, return the number of characters printed (not including the trailing NUL used to end output to strings) if there is enough space.

The formatting functions do not write more than *bufsiz* bytes (including the trailing NUL). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing NUL) which would have been written to the final string if enough space had been available. Thus, a return value of *bufsiz* or more means that the output was truncated.

Except for confd_val2str(), these functions will never return CONFD_ERR or any other negative value.

int **confd_ns_pp_value**(char **buf*, int *bufsiz*, const confd_value_t **v*, int *ns*);

This function is deprecated, but will remain for backward compatibility. It just calls confd_pp_value() - use confd_pp_value() directly, or confd_val2str() (see below), instead.

int **confd_pp_kpath**(char **buf*, int *bufsiz*, const confd_hkeypath_t **hkeypath*);

Utility function which pretty prints up to *bufsiz* characters into *buf*, giving a string representation of the path *hkeypath*. This will use the ConfD curly brace notation, i.e. "/servers/server{www}/ip". Requires that schema information is available to the library, see confd_types(3). Same return value as confd_pp_value().

int **confd_pp_kpath_len**(char **buf*, int *bufsiz*, const confd_hkeypath_t **hkeypath*, int *len*);

A variant of `confd_pp_kpath()` that prints only the first *`len`* elements of *`hkeypath`*.

`int` **`confd_format_keypath`**`(char *`*`buf`*`, int `*`bufsiz`*`, const char *`*`fmt`*`, ...);`

Several of the functions in confd_lib_maapi(3) and confd_lib_cdb(3) take a variable number of arguments which are then, similar to printf, used to generate the path passed to ConfD - see the PATHS section of confd_lib_cdb(3). This function takes the same arguments, but only formats the path as a string, writing at most *`bufsiz`* characters into *`buf`*. If the path is absolute and schema information is available to the library, key values referenced by a "%x" modifier will be printed according to their specific type, i.e. effectively using `confd_val2str()`, otherwise `confd_pp_value()` is used. Same return value as `confd_pp_value()`.

`int` **`confd_vformat_keypath`**`(char *`*`buf`*`, int `*`bufsiz`*`, const char *`*`fmt`*`, va_list `*`ap`*`);`

Does the same as `confd_format_keypath()`, but takes a single va_list argument instead of a variable number of arguments - i.e. similar to vprintf. Same return value as `confd_pp_value()`.

`char *`**`confd_xmltag2str`**`(u_int32_t `*`ns`*`, u_int32_t `*`xmltag`*`);`

This function is deprecated, but will remain for backward compatibility. It just calls `confd_hash2str()` - use `confd_hash2str()` directly instead, see below.

`int` **`confd_xpath_pp_kpath`**`(char *`*`buf`*`, int `*`bufsiz`*`, u_int32_t `*`ns`*`, const confd_hkeypath_t *`*`hkeypath`*`);`

Similar to `confd_pp_kpath()` except that the path is formatted as an XPath path, i.e. `"/servers:servers/server[name="www"]/ip"`. This function can also take the namespace integer as an argument. If `0` is passed as *`ns`*, the namespace is derived from the hkeypath. Requires that schema information is available to the library, see confd_types(3). Same return value as `confd_pp_value()`.

`int` **`confd_get_nslist`**`(struct confd_nsinfo **`*`listp`*`);`

Provides a list of the namespaces known to the library as an array of struct confd_nsinfo structures:

```
struct confd_nsinfo {
    const char *uri;
    const char *prefix;
    u_int32_t hash;
    const char *revision;
};
```

A pointer to the array is stored in *`*listp`*, and the function returns the number of elements in the array. The `revision` element in struct confd_nsinfo will give the revision for YANG modules that have a `revision` statement, otherwise it is NULL.

`char *`**`confd_ns2prefix`**`(u_int32_t `*`ns`*`);`

Returns a NUL-terminated string giving the namespace prefix for the namespace *`ns`*, if the namespace is known to the library - otherwise it returns NULL.

`char *`**`confd_hash2str`**`(u_int32_t `*`hash`*`);`

Returns a NUL-terminated string representing the node name given by *`hash`*, or NULL if the hash value is not found. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3) - otherwise it always returns NULL.

`u_int32_t` **`confd_str2hash`**`(const char *`*`str`*`);`

Returns the hash value representing the node name given by *str*, or 0 if the string is not found. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3) - otherwise it always returns 0.

```
struct confd_cs_node *confd_find_cs_root(int ns);
```

When schema information is available to the library, this function returns the root of the tree representaton of the namespace given by *ns*, i.e. a pointer to the struct confd_cs_node for the (first) toplevel node. For namespaces that are augmented into other namespaces such that they do not have a toplevel node, this function returns NULL - the nodes of such a namespace are found below the `augment` target node(s) in other tree(s). See confd_types(3).

```
struct confd_cs_node *confd_find_cs_node(const confd_hkeypath_t *hkey-
path, int len);
```

Utility function which finds the struct confd_cs_node corresponding to the *len* first elements of the hashed keypath. To make the search consider the full keypath, pass the *len* element from the confd_hkeypath_t structure (i.e. `mykeypath->len`). See confd_types(3).

```
struct confd_cs_node *confd_find_cs_node_child(const struct
confd_cs_node *parent, struct xml_tag xmltag);
```

Utility function which finds the struct confd_cs_node corresponding to the child node given as *xmltag*. See confd_types(3).

```
struct confd_cs_node *confd_cs_node_cd(const struct confd_cs_node
*start, const char *fmt, ...);
```

Utility function which finds the resulting struct confd_cs_node given an (optional) starting node and a (relative or absolute) string keypath. I.e. this function navigates the tree in a manner corresponding to `cdb_cd()`/`maapi_cd()`. Note however that the confd_cs_node tree does not have a node corresponding to "/". It is possible to pass *start* as `NULL`, in which case the path must be absolute (i.e. start with a "/").

Since the key values are not relevant for the tree navigation, the key elements can be omitted, i.e. a "tagpath" can be used - if present, key elements are ignored, whether given in the {...} form or the CDB-only [N] form. See confd_types(3).

If the path can not be found, `NULL` is returned, `confd_errno` is set to `CONFD_ERR_BADPATH`, and `confd_lasterr()` can be used to retrieve a string that describes the reason for the failure.

```
int confd_max_object_size(struct confd_cs_node *object);
```

Utility function which returns the maximum size (i.e. the needed length of the confd_value_t array) for an "object" retrieved by `cdb_get_object()`, `maapi_get_object()`, and corresponding multi-object functions. The *object* parameter is a pointer to the list or container confd_cs_node node for which we want to find the maximum size. See the description of `cdb_get_object()` in confd_lib_cdb(3) for usage examples.

```
struct confd_cs_node *confd_next_object_node(struct confd_cs_node *ob-
ject, struct confd_cs_node *cur, confd_value_t *value);
```

Utility function to allow navigation of the confd_cs_node schema tree in parallel with the confd_value_t array populated by `cdb_get_object()`, `maapi_get_object()`, and corresponding multi-object functions. The *object* parameter is a pointer to the list or container node as for `confd_max_object_size()`, the *cur* parameter is a pointer to the confd_cs_node node for the current value, and the *value* parameter is a pointer to the current value in the array. The function returns

a pointer to the confd_cs_node node for the next value in the array, or NULL when the complete object has been traversed. In the initial call for a given traversal, we must pass `object->children` for the *`cur`* parameter - this always points to the confd_cs_node node for the first value in the array. See the description of `cdb_get_object()` in confd_lib_cdb(3) for usage examples.

```
struct confd_type *confd_find_ns_type(u_int32_t nshash, const char
*name);
```

Returns a pointer to a type definition for the type named *`name`*, which is defined in the namespace identified by *`nshash`*, or NULL if the type could not be found. If *`nshash`* is 0, the type name will be looked up among the ConfD built-in types (i.e. the YANG built-in types, the types defined in the YANG "tailf-common" module, and the types defined in the "confd" and "xs" namespaces). The type definition pointer can be used with the `confd_val2str()` and `confd_str2val()` functions, see below. If *`nshash`* is not 0, the function requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3) - otherwise it returns NULL.

```
struct confd_type *confd_get_leaf_list_type(struct confd_cs_node
*node);
```

For a leaf-list node, the `type` field in the struct confd_cs_node_info (see confd_types(3)) identifies a "list type" for the leaf-list "itself". This function takes a pointer to the struct confd_cs_node for a leaf-list node as argument, and returns the type of the elements in the leaf-list, i.e. corresponding to the `type` substatement for the leaf-list in the YANG module. If called for a node that is not a leaf-list, it returns NULL and sets `confd_errno` to `CONFD_ERR_PROTOUSAGE`. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3) - otherwise it returns NULL and sets `confd_errno` to `CONFD_ERR_UNAVAILABLE`.

```
int confd_val2str(struct confd_type *type, const confd_value_t *val,
char *buf, int bufsiz);
```

Prints the string representation of *`val`* into *`buf`*, which has the length *`bufsiz`*, using type information from the data model. Returns the length of the string as described for `confd_pp_value()`, or CONFD_ERR if the value could not be converted (e.g. wrong type). The *`type`* pointer can be obtained either from the struct confd_cs_node corresponding to the leaf that *`val`* pertains to, or via the `confd_find_ns_type()` function above. The struct confd_cs_node can in turn be obtained by various combinations of the functions that operate on the confd_cs_node trees (see above), or by user-defined functions for navigating those trees. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3).

```
int confd_str2val(struct confd_type *type, const char *str,
confd_value_t *val);
```

Stores the value corresponding to the NUL-terminated string *`str`* in *`val`*, using type information from the data model. Returns CONFD_OK, or CONFD_ERR if the string could not be converted. See `confd_val2str()` for a description of the *`type`* argument. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3).

Note

When the resulting value is of one of the C_BUF, C_BINARY, C_LIST, C_OBJECTREF, C_OID, C_QNAME, or C_HEXSTR confd_value_t types, the library has allocated memory to hold the value. It is up to the user of this function to free the memory using `confd_free_value()`.

```
char *confd_val2str_ptr(struct confd_type *type, const confd_value_t
*val);
```

A variant of `confd_val2str()` that can be used only when the string representation is a constant, i.e. C_ENUM_VALUE values. In this case it returns a pointer to the string, otherwise NULL. See `confd_val2str()` for a description of the *`type`* argument. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3).

`int` **`confd_get_decimal64_fraction_digits`**`(struct confd_type *`*`type`*`);`

Utility function to obtain the value of the argument to the `fraction-digits` statement for a YANG decimal64 type. This is useful when we want to create a confd_value_t for such a type, since the `value` element must be scaled according to the fraction-digits value. The function returns the fraction-digits value, or 0 if the *`type`* argument does not refer to a decimal64 type. Requires that schema information has been loaded from the ConfD daemon into the library, see confd_types(3).

`int` **`confd_hkp_tagmatch`**`(struct xml_tag` *`tags[]`*`, int` *`tagslen`*`, confd_hkeypath_t *`*`hkp`*`);`

When checking the hkeypaths that get passed into each iteration in e.g. `cdb_diff_iterate()` we can either explicitly check the paths, or use this function to do the job. The *`tags`* array (typically statically initialized) specifies a tagpath to match against the hkeypath. See `cdb_diff_match()`. The function returns one of these values:

```
#define CONFD_HKP_MATCH_NONE 0
#define CONFD_HKP_MATCH_TAGS (1 << 0)
#define CONFD_HKP_MATCH_HKP  (1 << 1)
#define CONFD_HKP_MATCH_FULL (CONFD_HKP_MATCH_TAGS|CONFD_HKP_MATCH_HKP)
```

`CONFD_HKP_MATCH_TAGS` means that the whole tagpath was matched by the hkeypath, and `CONFD_HKP_MATCH_HKP` means that the whole hkeypath was matched by the tagpath.

`int` **`confd_hkp_prefix_tagmatch`**`(struct xml_tag` *`tags[]`*`, int` *`tagslen`*`, confd_hkeypath_t *`*`hkp`*`);`

A simplified version of `confd_hkp_tagmatch()` - it returns 1 if the tagpath matches a prefix of the hkeypath, i.e. it is equivalent to calling `confd_hkp_tagmatch()` and checking if the return value includes `CONFD_HKP_MATCH_TAGS`.

`int` **`confd_val_eq`**`(const confd_value_t *`*`v1`*`, const confd_value_t *`*`v2`*`);`

Utility function which compares two values. Returns positive value if equal, 0 otherwise.

`void` **`confd_fatal`**`(const char *`*`fmt`*`, ...);`

Utility function which formats a string, prints it to stderr and exits with exit code 1.

`void` **`confd_free_value`**`(confd_value_t *`*`v`*`);`

When we retrieve values via the CDB or MAAPI interfaces, or convert strings to values via `confd_str2val()`, and these values are of either of the types C_BUF, C_BINARY, C_QNAME, C_OBJECTREF, C_OID, C_LIST, or C_HEXSTR, the library has allocated memory to hold the values. This memory must be freed by the application when it is done with the value. This function frees memory for all confd_value_t types. Note that this function does not free the structure itself, only possible internal pointers inside the struct. Typically we use confd_value_t variables as automatic variables allocated on the stack. If the held value is of fixed size, e.g. integers, xmltags etc, the `confd_free_value()` function does nothing.

Note

Memory for values received as parameters to callback functions is always managed by the library - the application must *not* call `confd_free_value()` for those (on the other hand values of the types listed above that are received as parameters to a callback function must be copied if they are to persist beyond the callback invocation).

`confd_value_t *`**`confd_value_dup_to`**`(const confd_value_t *`*`v`*`, confd_value_t *`*`newv`*`);`

This function copies the contents of *`*v`* to *`*newv`*, allocating memory for the actual value for the types that need it. It returns *`newv`*, or NULL if allocation failed. The allocated memory (if any) can be freed with `confd_free_dup_to_value()`.

`void` **`confd_free_dup_to_value`**`(confd_value_t *`*`v`*`);`

Frees memory allocated by `confd_value_dup_to()`. Note this is not the same as `confd_free_value()`, since `confd_value_dup_to()` also allocates memory for values of type C_STR - such values are not freed by `confd_free_value()`.

`confd_value_t *`**`confd_value_dup`**`(const confd_value_t *`*`v`*`);`

This function allocates memory and duplicates *`*v`*, i.e. a confd_value_t struct is always allocated, memory for the actual value is also allocated for the types that need it. Returns a pointer to the new confd_value_t, or NULL if allocation failed. The allocated memory can be freed with `confd_free_dup_value()`.

`void` **`confd_free_dup_value`**`(confd_value_t *`*`v`*`);`

Frees memory allocated by `confd_value_dup()`. Note this is not the same as `confd_free_value()`, since `confd_value_dup()` also allocates the actual confd_value_t struct, and allocates memory for values of type C_STR - such values are not freed by `confd_free_value()`.

`confd_hkeypath_t *`**`confd_hkeypath_dup`**`(const confd_hkeypath_t *`*`src`*`);`

This function allocates memory and duplicates a confd_hkeypath_t.

`confd_hkeypath_t *`**`confd_hkeypath_dup_len`**`(const confd_hkeypath_t *`*`src`*`, int `*`len`*`);`

Like `confd_hkeypath_dup()`, but duplicates only the first *`len`* elements of the confd_hkeypath_t. I.e. the elements are shifted such that `v[0][0]` still refers to the last element.

`void` **`confd_free_hkeypath`**`(confd_hkeypath_t *`*`hkp`*`);`

This function will free memory allocated by e.g. `confd_hkeypath_dup()`.

`void` **`confd_free_authorization_info`**`(struct confd_authorization_info *`*`ainfo`*`);`

This function will free memory allocated by `maapi_get_authorization_info()`.

`int` **`confd_decrypt`**`(const char *`*`ciphertext`*`, int `*`len`*`, char *`*`output`*`);`

When data is read over the CDB interface, the MAAPI interface or received in event notifications, the data for the two builtin types tailf:des3-cbc-encrypted-string or tailf:aes-cfb-128-encrypted-string is encrypted.

This function decrypts *`len`* bytes of data from *`ciphertext`* and writes the clear text to the *`output`* pointer. The *`output`* pointer must point to an area that is at least *`len`* bytes long.

Note

One of the functions `confd_install_crypto_keys()` and `maapi_install_crypto_keys()` must have been called before `confd_decrypt()` can be used.

USER-DEFINED TYPES

It is possible to define new types, i.e. mappings between a textual representation and a confd_value_t representation that are not pre-defined in the ConfD daemon. Read more about this in the confd_types(3) manual page.

`int` **`confd_type_cb_init`**`(struct confd_type_cbs **`*`cbs`*`);`

This is the prototype for the function that a shared object implementing one or more user-defined types must provide. See confd_types(3).

`int` **`confd_register_ns_type`**`(u_int32_t` *`nshash`*`, const char *`*`name`*`, struct confd_type *`*`type`*`);`

This function can be used to register a user-defined type with the libconfd library, to make it possible for `confd_str2val()` and `confd_val2str()` to provide local string<->value translation in the application. See confd_types(3).

`int` **`confd_register_node_type`**`(struct confd_cs_node *`*`node`*`, struct confd_type *`*`type`*`);`

This function provides an alternate way to register a user-defined type with the libconfd library, in particular when the user-defined type is specified "inline" in a `leaf` or `leaf-list` statement. See confd_types(3).

CONFD STREAMS

Some functions in the ConfD lib stream data. Either from ConfD to the application of from the application to ConfD. The individual functions that use this feature will explicitly indicate that the data is passed over a `stream socket`.

`int` **`confd_stream_connect`**`(int` *`sock`*`, const struct sockaddr* `*`srv`*`, int` *`srv_sz`*`, int` *`id`*`, int` *`flags`*`);`

Connects a stream socket to ConfD. The *`id`* and the *`flags`* take different values depending on the usage scenario. This is indicated for each individual function that makes use of a stream socket.

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

MARSHALLING

In various distributed scenarios we may want to send confd_lib datatypes over the network. We have support to marshall and unmarshall some key datatypes.

`int` **`confd_serialize`**`(struct confd_serializable *`*`s`*`, unsigned char *`*`buf`*`, int` *`bufsz`*`, int *`*`bytes_written`*`, unsigned char **`*`allocated`*`);`

This function takes a confd_serializable struct as parameter. We have:

```
enum confd_serializable_type {
    CONFD_SERIAL_NONE      = 0,
    CONFD_SERIAL_VALUE_T   = 1,
    CONFD_SERIAL_HKEYPATH  = 2,
    CONFD_SERIAL_TAG_VALUE = 3
};
```

```
struct confd_serializable {
    enum confd_serializable_type type;
    union {
        confd_value_t *value;
        confd_hkeypath_t *hkp;
        confd_tag_value_t *tval;
    } u;
};
```

The structure must be populated with a valid type and also a value to be serialized. The serialized data will be written into the provided buffer. If the size of the buffer is insufficient, the function returns the required size as a positive integer. If the provided buffer is NULL, the function will allocate a buffer and it is the responsibility of the caller to free the buffer. The optionally allocated buffer is then returned in the output char ** parameter *allocated*. The function returns 0 on success and -1 on failures.

`int` **`confd_deserialize`**`(struct confd_deserializable *`*`s`*`, unsigned char *`*`buf`*`);`

This function takes a confd_deserializable struct as parameter. We have:

```
struct confd_deserializable {
    enum confd_serializable_type type;
    union {
        confd_value_t value;
        confd_hkeypath_t hkp;
        confd_tag_value_t tval;
    } u;
    void *internal;  // internal structure containing memory
                     // for the above datatypes to point _into_
                     // freed by a call to confd_deserialize_free()
};
```

This function is the reverse of `confd_serialize()`. It populates the provided confd_deserializable structure with a type indicator and a reproduced value of the correct type. The structure contains allocated memory that must subsequently be freed with `confd_deserialiaze()`.

`void` **`confd_deserialized_free`**`(struct confd_deserializable *`*`s`*`);`

A populated confd_deserializable struct contains allocated memory that must be freed. This function traverses a confd_deserializable struct as populated by the `confd_deserialize()` function and frees all allocated memory.

EXTENDED ERROR REPORTING

The data provider callback functions described in confd_lib_dp(3) can pass error information back to ConfD either as a simple string using `confd_xxx_seterr()`, or in a more structured/detailed form using the corresponding `confd_xxx_seterr_extended()` function. This form is also used when a CDB subscriber wishes to abort the current transaction with `cdb_sub_abort_trans()`, see confd_lib_cdb(3). There is also a set of `confd_xxx_seterr_extended_info()` functions and a

`cdb_sub_abort_trans_info()` function, that can alternatively be used if we want to provide contents for the NETCONF <error-info> element. The description below uses the functions for transaction callbacks as an example, but the other functions follow the same pattern:

`void` **`confd_trans_seterr_extended`**`(struct confd_trans_ctx *`*`tctx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, const char *`*`fmt`*`, ...);`

The function can be used also after a data provider callback has returned CONFD_DELAYED_RESPONSE, but in that case it must be followed by a call of `confd_delayed_reply_error()` (see confd_lib_dp(3)) with NULL for the *`errstr`* pointer.

One of the following values can be given for the *`code`* argument:

`CONFD_ERRCODE_IN_USE`
: Locking a data store was not possible because it was already locked.

`CONFD_ERRCODE_RESOURCE_DENIED`
: General resource unavailability, e.g. insufficient memory to carry out an operation.

`CONFD_ERRCODE_INCONSISTENT_VALUE`
: A request parameter had an unacceptable/invalid value

`CONFD_ERRCODE_ACCESS_DENIED`
: The request could not be fulfilled because authorization did not allow it. (No additional error information will be reported by the northbound agent, to avoid any security breach.)

`CONFD_ERRCODE_APPLICATION`
: Unspecified error.

`CONFD_ERRCODE_APPLICATION_INTERNAL`
: As CONFD_ERRCODE_APPLICATION, but the additional error information is only for logging/debugging, and should not be reported by northbound agents.

`CONFD_ERRCODE_DATA_MISSING`
: A request could not be completed because the relevant data model content does not exist.

`CONFD_ERRCODE_INTERRUPT`
: Processing of a request was terminated due to user interrupt - see the description of the `interrupt()` transaction callback in confd_lib_dp(3).

There is currently limited support for specifying one of a set of fixed error tags via *`apptag_ns`* and *`apptag_tag`*: *`apptag_ns`* should be 0, and *`apptag_tag`* can be either 0 or the hash value for a data model node.

The *`fmt`* and remaining arguments can specify an arbitrary string as for `confd_trans_seterr()`, but when used with one of the *`code`* values that has a specific meaning, it should only be given if it has some additional information - e.g. passing "In use" with CONFD_ERRCODE_IN_USE is not meaningful, and will typically result in duplicated information being reported by the northbound agent. If there is no additional information, just pass an empty string ("") for *`fmt`*.

A call of confd_trans_seterr(tctx, "string") is equivalent to confd_trans_seterr_extended(tctx, CONFD_ERRCODE_APPLICATION, 0, 0, "string").

When the extended error reporting is used, the northbound agents will, where possible, use the extended error information to give protocol-specific error reports to the managers, as described in the following

tables. (The CONFD_ERRCODE_INTERRUPT code does not have a mapping here, since these interfaces do not provide the possibility to interrupt a transaction.)

For SNMP, the *code* argument is mapped to SNMP ErrorStatus

code	**SNMP ErrorStatus**
CONFD_ERRCODE_IN_USE	resourceUnavailable
CONFD_ERRCODE_RESOURCE_DENIED	resourceUnavailable
CONFD_ERRCODE_INCONSISTENT_VALUE	inconsistentValue
CONFD_ERRCODE_ACCESS_DENIED	noAccess
CONFD_ERRCODE_APPLICATION	genErr
CONFD_ERRCODE_APPLICATION_INTERNAL	genErr
CONFD_ERRCODE_DATA_MISSING	inconsistentValue

For NETCONF the *code* argument is mapped to <error-tag>:

code	**NETCONF error-tag**
CONFD_ERRCODE_IN_USE	in-use
CONFD_ERRCODE_RESOURCE_DENIED	resource-denied
CONFD_ERRCODE_INCONSISTENT_VALUE	invalid-value
CONFD_ERRCODE_ACCESS_DENIED	access-denied
CONFD_ERRCODE_APPLICATION_	operation-failed
CONFD_ERRCODE_APPLICATION_INTERNAL	operation-failed
CONFD_ERRCODE_DATA_MISSING	data-missing

The tag specified by *apptag_ns*/*apptag_tag* will be reported as <error-app-tag>.

For MAAPI the *code* argument is mapped to confd_errno:

code	**confd_errno**
CONFD_ERRCODE_IN_USE	CONFD_ERR_INUSE
CONFD_ERRCODE_RESOURCE_DENIED	CONFD_ERR_RESOURCE_DENIED
CONFD_ERRCODE_INCONSISTENT_VALUE	CONFD_ERR_INCONSISTENT_VALUE
CONFD_ERRCODE_ACCESS_DENIED	CONFD_ERR_ACCESS_DENIED
CONFD_ERRCODE_APPLICATION	CONFD_ERR_EXTERNAL
CONFD_ERRCODE_APPLICATION_INTERNAL	CONFD_ERR_APPLICATION_INTERNAL
CONFD_ERRCODE_DATA_MISSING	CONFD_ERR_DATA_MISSING

The tag (if any) can be retrieved by calling

```
struct xml_tag *confd_last_error_apptag(void);
```

If no tag was provided by the callback (e.g. plain confd_trans_seterr() was used, or the error did not originate from a data provider callback at all), this function returns a pointer to a struct xml_tag with both the ns and the tag element set to 0.

In the CLI and Web UI a text string is produced through some combination of the *code* and the string given by *fmt, ...*.

`int` **`confd_trans_seterr_extended_info`**`(struct confd_trans_ctx *`*`tctx`*`, enum confd_errcode `*`code`*`, u_int32_t `*`apptag_ns`*`, u_int32_t `*`apptag_tag`*`, confd_tag_value_t *`*`error_info`*`, int `*`n`*`, const char *`*`fmt`*`, ...);`

This function can be used to provide structured error information in the same way as `confd_trans_seterr_extended()`, and additionally provide contents for the NETCONF <error-info> element. The *`error_info`* argument is an array of length *`n`*, populated as described for the Tagged Value Array format in the XML STRUCTURES section of the confd_types(3) manual page. The *`error_info`* information is discarded for other northbound agents than NETCONF.

The `tailf:error-info` statement (see tailf_yang_extensions(5)) must have been used in one or more YANG modules to declare the data nodes for <error-info>. As an example, we could have this `error-info` declaration:

```
module mod {
  namespace "http://tail-f.com/test/mod";
  prefix mod;

  import tailf-common {
    prefix tailf;
  }

  ...

  tailf:error-info {
    leaf severity {
      type enumeration {
        enum info;
        enum error;
        enum critical;
      }
    }
    container detail {
      leaf class {
        type uint8;
      }
      leaf code {
        type uint8;
      }
    }
  }

  ...

}
```

A call of `confd_trans_seterr_extended_info()` to populate the <error-info> could then look like this:

```
confd_tag_value_t error_info[10];
int i = 0;

CONFD_SET_TAG_ENUM_VALUE(&error_info[i],
                         mod_severity, mod_error);
CONFD_SET_TAG_NS(&error_info[i], mod__ns);          i++;
CONFD_SET_TAG_XMLBEGIN(&error_info[i],
                       mod_detail, mod__ns);        i++;
CONFD_SET_TAG_UINT8(&error_info[i], mod_class, 42); i++;
CONFD_SET_TAG_UINT8(&error_info[i], mod_code, 17);  i++;
```

```
CONFD_SET_TAG_XMLEND(&error_info[i],
                     mod_detail, mod__ns);          i++;
OK(confd_trans_seterr_extended_info(tctx, CONFD_ERRCODE_APPLICATION,
                                    0, 0, error_info, i,
                                    "Operation failed"));
```

Note

The toplevel elements in the confd_tag_value_t array *must* have the `ns` element of the struct xml_tag set. The `CONFD_SET_TAG_XMLBEGIN()` macro will set this element, but for toplevel leaf elements the `CONFD_SET_TAG_NS()` macro needs to be used, as shown above.

The <error-info> section resulting from the above would look like this:

```
    <error-info>
      ...
      <severity xmlns="http://tail-f.com/test/mod">error</severity>
      <detail xmlns="http://tail-f.com/test/mod">
        <class>42</class>
        <code>17</code>
      </detail>
    </error-info>
```

ERRORS

All functions in `libconfd` signal errors through the return of the #defined CONFD_ERR - which has the value -1 - or alternatively CONFD_EOF (-2) which means that ConfD closed its end of the socket.

Data provider callbacks (see confd_lib_dp(3)) can also signal errors by returning CONFD_ERR from the callback. This can be done for all different kinds of callbacks. It is possible to to provide additional error information from one of these callbacks by using one of the functions:

`confd_trans_seterr()`, `confd_trans_seterr_extended()`, `confd_trans_seterr_extended_info()` — For transaction callbacks

`confd_db_seterr()`, `confd_db_seterr_extended()`, `confd_db_seterr_extended_info()` — For db callbacks

`confd_action_seterr()`, `confd_action_seterr_extended()`, `confd_action_seterr_extended_info()` — For action callbacks

`confd_notification_seterr()`, `confd_notification_seterr_extended()`, `confd_notification_seterr_extended_info()` — For notification callbacks

CDB two phase subscribers (see confd_lib_cdb(3)) can also provide error information when `cdb_read_subscription_socket2()` has returned with type set to `CDB_SUB_PREPARE`, using one of the functions `cdb_sub_abort_trans()` and `cdb_sub_abort_trans_info()`.

Whenever CONFD_ERR is returned from any API function in `libconfd` it is possible to obtain additional information on the error through the symbol `confd_errno`. Additionally there may be an error text associated with the error. A call to the function

```
char *confd_lasterr(void);
```

returns a string which contains additional textual information on the error. Furthermore, the function

```
char *confd_strerror(int code);
```

returns a string which describes a particular error code. When one of the The following error codes are available:

CONFD_ERR_NOEXISTS (1)
Typically we tried to read a value through CDB or MAAPI which does not exist.

CONFD_ERR_ALREADY_EXISTS (2)
We tried to create something which already exists.

CONFD_ERR_ACCESS_DENIED (3)
Access to an object was denied due to AAA authorization rules.

CONFD_ERR_NOT_WRITABLE (4)
We tried to write an object which is not writable.

CONFD_ERR_BADTYPE (5)
We tried to create or write an object which is specified to have another type (see confd_types(3)) than the one we provided.

CONFD_ERR_NOTCREATABLE (6)
We tried to create an object which is not possible to create.

CONFD_ERR_NOTDELETABLE (7)
We tried to delete an object which is not possible to delete.

CONFD_ERR_BADPATH (8)
We provided a bad path in any of the printf style functions which take a variable number of arguments.

CONFD_ERR_NOSTACK (9)
We tried to pop without a preceding push.

CONFD_ERR_LOCKED (10)
We tried to lock something which is already locked.

CONFD_ERR_INUSE (11)
We tried to commit while someone else holds a lock.

CONFD_ERR_NOTSET (12)
A mandatory leaf does not have a value, either because it has been deleted, or not set after a create.

CONFD_ERR_NON_UNIQUE (13)
A group of leafs specified with the `unique` statement are not unique.

CONFD_ERR_BAD_KEYREF (14)
Dangling pointer.

CONFD_ERR_TOO_FEW_ELEMS (15)
A `min-elements` violation. A node has fewer elements or entries than specified with `min-elements`.

CONFD_ERR_TOO_MANY_ELEMS (16)
A `max-elements` violation. A node has fewer elements or entries than specified with `max-elements`.

CONFD_ERR_BADSTATE (17)
Some function, such as the MAAPI commit functions that require several functions to be called in a specific order, was called out of order.

CONFD_ERR_INTERNAL (18)
An internal error. This normally indicates a bug in ConfD or libconfd (if nothing else the lack of a better error code), please report it to Tail-f support.

CONFD_ERR_EXTERNAL (19)
All errors that originate in user code.

CONFD_ERR_MALLOC (20)
Failed to allocate memory.

CONFD_ERR_PROTOUSAGE (21)
Usage of API functions or callbacks was wrong. It typically means that we invoke a function when we shouldn't. For example if we invoke the `confd_data_reply_next_key()` in a `get_elem()` callback we get this error.

CONFD_ERR_NOSESSION (22)
A session must be established prior to executing the function.

CONFD_ERR_TOOMANYTRANS (23)
A new MAAPI transaction was rejected since the transaction limit threshold was reached.

CONFD_ERR_OS (24)
An error occurred in a call to some operating system function, such as `write()`. The proper errno from libc should then be read and used as failure indicator.

CONFD_ERR_HA_CONNECT (25)
Failed to connect to a remote HA node.

CONFD_ERR_HA_CLOSED (26)
A remote HA node closed its connection to us, or there was a timeout waiting for a sync response from the master during a call of `confd_ha_beslave()`.

CONFD_ERR_HA_BADFXS (27)
A remote HA node had a different set of fxs files compared to us. It could also be that the set is the same, but the version of some fxs file is different.

CONFD_ERR_HA_BADTOKEN (28)
A remote HA node has a different token than us.

CONFD_ERR_HA_BADNAME (29)
A remote ha node has a different name than the name we think it has.

CONFD_ERR_HA_BIND (30)
Failed to bind the ha socket for incoming HA connects.

CONFD_ERR_HA_NOTICK (31)
A remote HA node failed to produce the interval live ticks.

CONFD_ERR_VALIDATION_WARNING (32)
`maapi_validate()` returned warnings.

CONFD_ERR_SUBAGENT_DOWN (33)
An operation towards a mounted NETCONF subagent failed due to the subagent not being up.

`CONFD_ERR_LIB_NOT_INITIALIZED` (34)
The confd library has not been properly initialized by a call to `confd_init()`.

`CONFD_ERR_TOO_MANY_SESSIONS` (35)
Maximum number of sessions reached.

`CONFD_ERR_BAD_CONFIG` (36)
An error in a configuration.

`CONFD_ERR_RESOURCE_DENIED` (37)
A data provider callback returned CONFD_ERRCODE_RESOURCE_DENIED (see EXTENDED ERROR REPORTING above).

`CONFD_ERR_INCONSISTENT_VALUE` (38)
A data provider callback returned CONFD_ERRCODE_INCONSISTENT_VALUE (see EXTENDED ERROR REPORTING above).

`CONFD_ERR_APPLICATION_INTERNAL` (39)
A data provider callback returned CONFD_ERRCODE_APPLICATION_INTERNAL (see EXTENDED ERROR REPORTING above).

`CONFD_ERR_UNSET_CHOICE` (40)
No `case` has been selected for a mandatory `choice` statement.

`CONFD_ERR_MUST_FAILED` (41)
A `must` constraint is not satisfied.

`CONFD_ERR_MISSING_INSTANCE` (42)
The value of an `instance-identifier` leaf with `require-instance true` does not specify an existing instance.

`CONFD_ERR_INVALID_INSTANCE` (43)
The value of an `instance-identifier` leaf does not conform to the specified path filters.

`CONFD_ERR_UNAVAILABLE` (44)
We tried to use some unavailable functionality, e.g. get/set attributes on an operational data element.

`CONFD_ERR_EOF` (45)
This value is used when a function returns CONFD_EOF. Thus it is not strictly necessary to check whether the return value is CONFD_ERR or CONFD_EOF - if the function should return CONFD_OK on success, but the return value is something else, the reason can always be found via confd_errno.

`CONFD_ERR_NOTMOVABLE` (46)
We tried to move an object which is not possible to move.

`CONFD_ERR_HA_WITH_UPGRADE` (47)
We tried to perform an in-service data model upgrade on a HA node that was either a master with slaves or a slave, or we tried to make the node a HA slave while an in-service data model upgrade was in progress.

`CONFD_ERR_TIMEOUT` (48)
An operation did not complete within the specified timeout.

`CONFD_ERR_ABORTED` (49)
An operation was aborted.

`CONFD_ERR_XPATH` (50)
Compilation or evaluation of an XPath expression failed.

`CONFD_ERR_NOT_IMPLEMENTED` (51)
A request was made for an operation that wasn't implemented. This will typically occur if an application uses a version of `libconfd` that is more recent than the version of the ConfD daemon, and a CDB or MAAPI function is used that is only implemented in the library version.

`CONFD_ERR_HA_BADVSN` (52)
A remote HA node had an incompatible protocol version.

`CONFD_ERR_POLICY_FAILED` (53)
A user-defined policy expression evaluated to false.

`CONFD_ERR_POLICY_COMPILATION_FAILED` (54)
A user-defined policy XPath expression could not be compiled.

`CONFD_ERR_POLICY_EVALUATION_FAILED` (55)
A user-defined policy expression failed XPath evaluation.

`NCS_ERR_CONNECTION_REFUSED` (56)
NCS failed to connect to a device.

`CONFD_ERR_START_FAILED` (57)
ConfD daemon failed to proceed to next start-phase.

`CONFD_ERR_DATA_MISSING` (58)
A data provider callback returned CONFD_ERRCODE_DATA_MISSING (see EXTENDED ERROR REPORTING above).

`CONFD_ERR_CLI_CMD` (59)
Execution of a CLI command failed.

`CONFD_ERR_UPGRADE_IN_PROGRESS` (60)
A request was made for an operation that is not allowed when in-service data model upgrade is in progress.

`CONFD_ERR_NOTRANS` (61)
An invalid transaction handle was passed to a MAAPI function - i.e. the handle did not refer to a transaction that was either started on, or attached to, the MAAPI socket.

`NCS_ERR_SERVICE_CONFLICT` (62)
An NCS service invocation running outside the transaction lock modified data that was also modified by a service invocation in another transaction.

MISCELLANEOUS

The library will always set the default signal handler for SIGPIPE to be SIG_IGN. All libconfd APIs are socket based and the library must be able to detect failed write operations in a controlled manner.

The include file `confd_lib.h` includes `assert.h` and uses assert macros in the specialized `CONFD_GET_XXX()` macros. If the behavior of assert is not wanted in a production environment, we can define NDEBUG before including `confd_lib.h` (or `confd.h`), see assert(3). Alternatively we can define a `CONFD_ASSERT()` macro before including `confd_lib.h`. The assert macros are invoked via `CONFD_ASSERT()`, which is defined by:

```
#ifndef CONFD_ASSERT
#define CONFD_ASSERT(E) assert(E)
#endif
```

I.e. by defining a different version of `CONFD_ASSERT()`, we can get our own error handler invoked instead of assert(3), for example:

```
void log_error(char *file, int line, char *expr);

#define CONFD_ASSERT(E) \
            ((E) ? (void)0 : log_error(__FILE__, __LINE__, #E))

#include <confd_lib.h>
```

SYSLOG AND DEBUG

When developing applications with `libconfd` we always need to indicate to the library which verbosity level should be used by the library. There are three different levels to choose from: CONFD_SILENT where the library never writes anything, never, CONFD_DEBUG where the library reports all errors and finally CONFD_TRACE where the library traces the execution and invocations of all the various callback functions.

There are two different destinations for all library printouts. When we call `confd_init()`, we always need to supply a FILE* stream which should be used for all printouts. This parameter can be set to NULL if we never want any FILE* printouts to occur.

The second destination is syslog, i.e. the library will syslog if told to. This is controlled by the global integer variable `confd_lib_use_syslog`. If we set this variable to `1`, `libconfd` will syslog all output. If we set it to `0` the library will not syslog. It is the responsibility of the application to (optionally) call `openlog()` before initializing the ConfD library. The default value is `0`.

There also exists a hook point at which a library user can install their own printer. This done by assigning to a global variable `confd_user_log_hook`, as in:

```
void mylogger(int syslogprio, const char *fmt, va_list ap) {
    char buf[BUFSIZ];
    sprintf(buf, "MYLOG:(%d) ", syslogprio); strcat(buf, fmt);
    vfprintf(stderr, buf, ap);
}

confd_user_log_hook = mylogger;
```

The *`syslogprio`* is LOG_ERR or LOG_CRIT for error messages, and LOG_DEBUG for trace messages, see the description of `confd_init()`.

Thus a good combination of values in a target environment is to set the FILE* handle to NULL and `confd_lib_use_syslog` to `1`. This way we do not get the overhead of file logging and at the same time get all errors reported to syslog.

SEE ALSO

confd.conf(5) - ConfD daemon configuration file format

The ConfD User Guide

Name

confd_lib_maapi — MAAPI (Management Agent API). A library for connecting to ConfD with a read/write interface inside transactions.

Synopsis

#include <confd_lib.h> #include <confd_maapi.h>

```
int maapi_connect(int sock, const struct sockaddr* srv, int srv_sz);

int maapi_load_schemas(int sock);

int maapi_load_schemas_list(int sock, int flags, const u_int32_t
*nshash, const int *nsflags, int num_ns);

int maapi_close(int sock);

int maapi_start_user_session(int sock, const char *username, const char
*context, const char **groups, int numgroups, const struct confd_ip
*src_addr, enum confd_proto prot);

int maapi_start_user_session2(int sock, const char *username, const char
*context, const char **groups, int numgroups, const struct confd_ip
*src_addr, int src_port, enum confd_proto prot);

int maapi_end_user_session(int sock);

int maapi_kill_user_session(int sock, int usessid);

int maapi_get_user_sessions(int sock, int res[], int n);

int maapi_get_user_session(int sock, int usessid, struct confd_user_info
*us);

int maapi_get_my_user_session_id(int sock);

int maapi_set_user_session(int sock, int usessid);

int maapi_get_user_session_identification(int sock, int usessid, struct
confd_user_identification *uident);

int maapi_get_user_session_opaque(int sock, int usessid, char **opaque);

int maapi_get_authorization_info(int sock, int usessid, struct
confd_authorization_info **ainfo);

int maapi_lock(int sock, enum confd_dbname name);

int maapi_unlock(int sock, enum confd_dbname name);

int maapi_is_lock_set(int sock, enum confd_dbname name);

int maapi_lock_partial(int sock, enum confd_dbname name, char *xpaths[],
int nxpaths, int *lockid);

int maapi_unlock_partial(int sock, int lockid);
```

int **maapi_candidate_validate**(int *sock*);

int **maapi_delete_config**(int *sock*, enum confd_dbname *name*);

int **maapi_candidate_commit**(int *sock*);

int **maapi_candidate_commit_persistent**(int *sock*, const char **persist_id*);

int **maapi_candidate_commit_persistent_flags**(int *sock*, const char **persist_id*, int *flags*);

int **maapi_candidate_confirmed_commit**(int *sock*, int *timeoutsecs*);

int **maapi_candidate_confirmed_commit_persistent**(int *sock*, int *timeoutsecs*, const char **persist*, const char **persist_id*);

int **maapi_candidate_confirmed_commit_persistent_flags**(int *sock*, int *timeoutsecs*, const char **persist*, const char **persist_id*, int *flags*);

int **maapi_candidate_abort_commit**(int *sock*);

int **maapi_candidate_abort_commit_persistent**(int *sock*, const char **persist_id*);

int **maapi_candidate_reset**(int *sock*);

int **maapi_confirmed_commit_in_progress**(int *sock*);

int **maapi_copy_running_to_startup**(int *sock*);

int **maapi_is_running_modified**(int *sock*);

int **maapi_is_candidate_modified**(int *sock*);

int **maapi_start_trans**(int *sock*, enum confd_dbname *name*, enum confd_trans_mode *readwrite*);

int **maapi_start_trans2**(int *sock*, enum confd_dbname *name*, enum confd_trans_mode *readwrite*, int *usid*);

int **maapi_start_trans_flags**(int *sock*, enum confd_dbname *name*, enum confd_trans_mode *readwrite*, int *usid*, int *flags*);

int **maapi_start_trans_in_trans**(int *sock*, enum confd_trans_mode *readwrite*, int *usid*, int *thandle*);

int **maapi_finish_trans**(int *sock*, int *thandle*);

int **maapi_validate_trans**(int *sock*, int *thandle*, int *unlock*, int *forcevalidation*);

int **maapi_prepare_trans**(int *sock*, int *thandle*);

int **maapi_prepare_trans_flags**(int *sock*, int *thandle*, int *flags*);

int **maapi_commit_trans**(int *sock*, int *thandle*);

int **maapi_abort_trans**(int *sock*, int *thandle*);

int **maapi_apply_trans**(int *sock*, int *thandle*, int *keepopen*);

int **maapi_apply_trans_flags**(int *sock*, int *thandle*, int *keepopen*, int *flags*);

int **maapi_commit_queue_result**(int *sock*, int *thandle*, int *timeoutsecs*, struct ncs_commit_queue_result **result*);

int **maapi_set_namespace**(int *sock*, int *thandle*, int *hashed_ns*);

int **maapi_cd**(int *sock*, int *thandle*, const char **fmt*, ...);

int **maapi_pushd**(int *sock*, int *thandle*, const char **fmt*, ...);

int **maapi_popd**(int *sock*, int *thandle*);

int **maapi_getcwd**(int *sock*, int *thandle*, size_t *strsz*, char **curdir*);

int **maapi_getcwd_kpath**(int *sock*, int *thandle*, confd_hkeypath_t ***kp*);

int **maapi_exists**(int *sock*, int *thandle*, const char **fmt*, ...);

int **maapi_num_instances**(int *sock*, int *thandle*, const char **fmt*, ...);

int **maapi_get_elem**(int *sock*, int *thandle*, confd_value_t **v*, const char **fmt*, ...);

int **maapi_get_int8_elem**(int *sock*, int *thandle*, int8_t **rval*, const char **fmt*, ...);

int **maapi_get_int16_elem**(int *sock*, int *thandle*, int16_t **rval*, const char **fmt*, ...);

int **maapi_get_int32_elem**(int *sock*, int *thandle*, int32_t **rval*, const char **fmt*, ...);

int **maapi_get_int64_elem**(int *sock*, int *thandle*, int64_t **rval*, const char **fmt*, ...);

int **maapi_get_u_int8_elem**(int *sock*, int *thandle*, u_int8_t **rval*, const char **fmt*, ...);

int **maapi_get_u_int16_elem**(int *sock*, int *thandle*, u_int16_t **rval*, const char **fmt*, ...);

int **maapi_get_u_int32_elem**(int *sock*, int *thandle*, u_int32_t **rval*, const char **fmt*, ...);

int **maapi_get_u_int64_elem**(int *sock*, int *thandle*, u_int64_t **rval*, const char **fmt*, ...);

int **maapi_get_ipv4_elem**(int *sock*, int *thandle*, struct in_addr **rval*, const char **fmt*, ...);

int **maapi_get_ipv6_elem**(int *sock*, int *thandle*, struct in6_addr **rval*, const char **fmt*, ...);

int **maapi_get_double_elem**(int *sock*, int *thandle*, double **rval*, const char **fmt*, ...);

int **maapi_get_bool_elem**(int *sock*, int *thandle*, int **rval*, const char **fmt*, ...);

int **maapi_get_datetime_elem**(int *sock*, int *thandle*, struct confd_datetime **rval*, const char **fmt*, ...);

int **maapi_get_date_elem**(int *sock*, int *thandle*, struct confd_date **rval*, const char **fmt*, ...);

int **maapi_get_time_elem**(int *sock*, int *thandle*, struct confd_time **rval*, const char **fmt*, ...);

int **maapi_get_duration_elem**(int *sock*, int *thandle*, struct confd_duration **rval*, const char **fmt*, ...);

int **maapi_get_enum_value_elem**(int *sock*, int *thandle*, int32_t **rval*, const char **fmt*, ...);

int **maapi_get_bit32_elem**(int *sock*, int *thandle*, u_int32_t **rval*, const char **fmt*, ...);

int **maapi_get_bit64_elem**(int *sock*, int *thandle*, u_int64_t **rval*, const char **fmt*, ...);

int **maapi_get_objectref_elem**(int *sock*, int *thandle*, confd_hkeypath_t ***rval*, const char **fmt*, ...);

int **maapi_get_oid_elem**(int *sock*, int *thandle*, struct confd_snmp_oid ***rval*, const char **fmt*, ...);

int **maapi_get_buf_elem**(int *sock*, int *thandle*, unsigned char ***rval*, int **bufsiz*, const char **fmt*, ...);

int **maapi_get_str_elem**(int *sock*, int *thandle*, char **buf*, int *n*, const char **fmt*, ...);

int **maapi_get_binary_elem**(int *sock*, int *thandle*, unsigned char ***rval*, int **bufsiz*, const char **fmt*, ...);

int **maapi_get_hexstr_elem**(int *sock*, int *thandle*, unsigned char ***rval*, int **bufsiz*, const char **fmt*, ...);

int **maapi_get_qname_elem**(int *sock*, int *thandle*, unsigned char ***prefix*, int **prefixsz*, unsigned char ***name*, int **namesz*, const char **fmt*, ...);

int **maapi_get_list_elem**(int *sock*, int *thandle*, confd_value_t ***values*, int **n*, const char **fmt*, ...);

int **maapi_get_ipv4prefix_elem**(int *sock*, int *thandle*, struct confd_ipv4_prefix **rval*, const char **fmt*, ...);

int **maapi_get_ipv6prefix_elem**(int *sock*, int *thandle*, struct confd_ipv6_prefix **rval*, const char **fmt*, ...);

int **maapi_get_decimal64_elem**(int *sock*, int *thandle*, struct confd_decimal64 *_rval_, const char *_fmt_, ...);

int **maapi_get_identityref_elem**(int *sock*, int *thandle*, struct confd_identityref *_rval_, const char *_fmt_, ...);

int **maapi_get_ipv4_and_plen_elem**(int *sock*, int *thandle*, struct confd_ipv4_prefix *_rval_, const char *_fmt_, ...);

int **maapi_get_ipv6_and_plen_elem**(int *sock*, int *thandle*, struct confd_ipv6_prefix *_rval_, const char *_fmt_, ...);

int **maapi_get_dquad_elem**(int *sock*, int *thandle*, struct confd_dotted_quad *_rval_, const char *_fmt_, ...);

int **maapi_vget_elem**(int *sock*, int *thandle*, confd_value_t *_v_, const char *_fmt_, va_list *_args_);

int **maapi_init_cursor**(int *sock*, int *thandle*, struct maapi_cursor *_mc_, const char *_fmt_, ...);

int **maapi_get_next**(struct maapi_cursor *_mc_);

int **maapi_find_next**(struct maapi_cursor *_mc_, enum confd_find_next_type *_type_, confd_value_t *_inkeys_, int *_n_inkeys_);

void **maapi_destroy_cursor**(struct maapi_cursor *_mc_);

int **maapi_set_elem**(int *sock*, int *thandle*, confd_value_t *_v_, const char *_fmt_, ...);

int **maapi_set_elem2**(int *sock*, int *thandle*, const char *_strval_, const char *_fmt_, ...);

int **maapi_vset_elem**(int *sock*, int *thandle*, confd_value_t *_v_, const char *_fmt_, va_list *_args_);

int **maapi_create**(int *sock*, int *thandle*, const char *_fmt_, ...);

int **maapi_delete**(int *sock*, int *thandle*, const char *_fmt_, ...);

int **maapi_get_object**(int *sock*, int *thandle*, confd_value_t *_values_, int *_n_, const char *_fmt_, ...);

int **maapi_get_objects**(struct maapi_cursor *_mc_, confd_value_t *_values_, int *_n_, int *_nobj_);

int **maapi_get_values**(int *sock*, int *thandle*, confd_tag_value_t *_values_, int *_n_, const char *_fmt_, ...);

int **maapi_set_object**(int *sock*, int *thandle*, const confd_value_t *_values_, int *_n_, const char *_fmt_, ...);

int **maapi_set_values**(int *sock*, int *thandle*, const confd_tag_value_t *_values_, int *_n_, const char *_fmt_, ...);

int **maapi_get_case**(int *sock*, int *thandle*, const char *_choice_, confd_value_t *_rcase_, const char *_fmt_, ...);

int **maapi_get_attrs**(int *sock*, int *thandle*, u_int32_t **attrs*, int *num_attrs*, confd_attr_value_t ***attr_vals*, int **num_vals*, const char **fmt*, ...);

int **maapi_set_attr**(int *sock*, int *thandle*, u_int32_t *attr*, confd_value_t **v*, const char **fmt*, ...);

int **maapi_delete_all**(int *sock*, int *thandle*, enum maapi_delete_how *how*);

int **maapi_revert**(int *sock*, int *thandle*);

int **maapi_set_flags**(int *sock*, int *thandle*, int *flags*);

int **maapi_set_delayed_when**(int *sock*, int *thandle*, int *on*);

int **maapi_copy**(int *sock*, int *from_thandle*, int *to_thandle*);

int **maapi_copy_path**(int *sock*, int *from_thandle*, int *to_thandle*, const char **fmt*, ...);

int **maapi_copy_tree**(int *sock*, int *thandle*, const char **from*, const char **tofmt*, ...);

int **maapi_insert**(int *sock*, int *thandle*, const char **fmt*, ...);

int **maapi_move**(int *sock*, int *thandle*, confd_value_t* *tokey*, int *n*, const char **fmt*, ...);

int **maapi_move_ordered**(int *sock*, int *thandle*, enum maapi_move_where *where*, confd_value_t* *tokey*, int *n*, const char **fmt*, ...);

int **maapi_shared_create**(int *sock*, int *thandle*, int *flags*, const char **fmt*, ...);

int **maapi_shared_set_elem**(int *sock*, int *thandle*, confd_value_t **v*, int *flags*, const char **fmt*, ...);

int **maapi_shared_set_elem2**(int *sock*, int *thandle*, const char **strval*, int *flags*, const char **fmt*, ...);

int **maapi_shared_insert**(int *sock*, int *thandle*, int *flags*, const char **fmt*, ...);

int **maapi_shared_copy_tree**(int *sock*, int *thandle*, int *flags*, const char **from*, const char **tofmt*, ...);

int **maapi_ncs_apply_template**(int *sock*, int *thandle*, char **template_name*, const struct ncs_name_value **variables*, int *num_variables*, int *flags*, const char **rootfmt*, ...);

int **maapi_shared_ncs_apply_template**(int *sock*, int *thandle*, char **template_name*, const struct ncs_name_value **variables*, int *num_variables*, int *flags*, const char **rootfmt*, ...);

int **maapi_ncs_get_templates**(int *sock*, char ****templates*, int **num_templates*);

int **maapi_authenticate**(int *sock*, const char **user*, const char **pass*, char **groups[]*, int *n*);

int **maapi_authenticate2**(int *sock*, const char **user*, const char **pass*, const struct confd_ip **src_addr*, int *src_port*, const char **context*, enum confd_proto *prot*, char **groups[]*, int *n*);

int **maapi_attach**(int *sock*, int *hashed_ns*, struct confd_trans_ctx **ctx*);

int **maapi_attach2**(int *sock*, int *hashed_ns*, int *usid*, int *thandle*);

int **maapi_attach_init**(int *sock*, int **thandle*);

int **maapi_detach**(int *sock*, struct confd_trans_ctx **ctx*);

int **maapi_detach2**(int *sock*, int *thandle*);

int **maapi_diff_iterate**(int *sock*, int *thandle*, enum maapi_iter_ret (**iter*)(confd_hkeypath_t **kp*, enum maapi_iter_op *op*, confd_value_t **oldv*, confd_value_t **newv*, void **state*), int *flags*, void **init_state*);

int **maapi_keypath_diff_iterate**(int *sock*, int *thandle*, enum maapi_iter_ret (**iter*)(confd_hkeypath_t **kp*, enum maapi_iter_op *op*, confd_value_t **oldv*, confd_value_t **newv*, void **state*), int *flags*, void **initstate*, const char **fmtpath*, ...);

int **maapi_iterate**(int *sock*, int *thandle*, enum maapi_iter_ret (**iter*)(confd_hkeypath_t **kp*, confd_value_t **v*, confd_attr_value_t **attr_vals*, int *num_attr_vals*, void **state*), int *flags*, void **initstate*, const char **fmtpath*, ...);

int **maapi_get_running_db_status**(int *sock*);

int **maapi_set_running_db_status**(int *sock*, int *status*);

int **maapi_list_rollbacks**(int *sock*, struct maapi_rollback **rp*, int **rp_size*);

int **maapi_load_rollback**(int *sock*, int *thandle*, int *rollback_num*);

int **maapi_request_action**(int *sock*, confd_tag_value_t **params*, int *nparams*, confd_tag_value_t ***values*, int **nvalues*, int *hashed_ns*, const char **fmt*, ...);

int **maapi_request_action_th**(int *sock*, int *thandle*, confd_tag_value_t **params*, int *nparams*, confd_tag_value_t ***values*, int **nvalues*, const char **fmt*, ...);

int **maapi_request_action_str_th**(int *sock*, int *thandle*, char ***output*, const char **cmd_fmt*, const char **path_fmt*, ...);

int **maapi_xpath2kpath**(int *sock*, const char **xpath*, confd_hkeypath_t ***hkp*);

int **maapi_user_message**(int *sock*, const char **to*, const char **message*, const char **sender*);

int **maapi_sys_message**(int *sock*, const char **to*, const char **message*);

int **maapi_prio_message**(int *sock*, const char **to*, const char **message*);

int **maapi_cli_diff_cmd**(int *sock*, int *thandle*, int *thandle_old*, char **res*, int *size*, int *flags*, const char **fmt*, ...);

int **maapi_cli_accounting**(int *sock*, const char **user*, const int *usid*, const char **cmdstr*);

int **maapi_cli_path_cmd**(int *sock*, int *thandle*, char **res*, int *size*, int *flags*, const char **fmt*, ...);

int **maapi_cli_cmd_to_path**(int *sock*, const char **line*, char **ns*, int *nsize*, char **path*, int *psize*);

int **maapi_cli_cmd_to_path2**(int *sock*, int *thandle*, const char **line*, char **ns*, int *nsize*, char **path*, int *psize*);

int **maapi_cli_prompt**(int *sock*, int *usess*, const char **prompt*, int *echo*, char **res*, int *size*);

int **maapi_cli_prompt2**(int *sock*, int *usess*, const char **prompt*, int *echo*, int *timeout*, char **res*, int *size*);

int **maapi_cli_prompt_oneof**(int *sock*, int *usess*, const char **prompt*, char ***choice*, int *count*, char **res*, int *size*);

int **maapi_cli_prompt_oneof2**(int *sock*, int *usess*, const char **prompt*, char ***choice*, int *count*, int *timeout*, char **res*, int *size*);

int **maapi_cli_read_eof**(int *sock*, int *usess*, int *echo*, char **res*, int *size*);

int **maapi_cli_read_eof2**(int *sock*, int *usess*, int *echo*, int *timeout*, char **res*, int *size*);

int **maapi_cli_write**(int *sock*, int *usess*, const char **buf*, int *size*);

int **maapi_cli_cmd**(int *sock*, int *usess*, const char **buf*, int *size*);

int **maapi_cli_cmd2**(int *sock*, int *usess*, const char **buf*, int *size*, int *flags*);

int **maapi_cli_cmd3**(int *sock*, int *usess*, const char **buf*, int *size*, int *flags*, const char **unhide*, int *usize*);

int **maapi_cli_cmd4**(int *sock*, int *usess*, const char **buf*, int *size*, int *flags*, char ***unhide*, int *usize*);

int **maapi_cli_cmd_io**(int *sock*, int *usess*, const char **buf*, int *size*, int *flags*, const char **unhide*, int *usize*);

int **maapi_cli_cmd_io2**(int *sock*, int *usess*, const char **buf*, int *size*, int *flags*, char ***unhide*, int *usize*);

int **maapi_cli_cmd_io_result**(int *sock*, int *id*);

int **maapi_cli_printf**(int *sock*, int *usess*, const char **fmt*, ...);

int **maapi_cli_vprintf**(int *sock*, int *usess*, const char **fmt*, va_list *args*);

int **maapi_cli_set**(int *sock*, int *usess*, const char **opt*, const char **value*);

int **maapi_cli_get**(int *sock*, int *usess*, const char **opt*, char **res*, int *size*);

int **maapi_set_readonly_mode**(int *sock*, int *flag*);

int **maapi_disconnect_remote**(int *sock*, const char **address*);

int **maapi_disconnect_sockets**(int *sock*, int **sockets*, int *nsocks*);

int **maapi_save_config**(int *sock*, int *thandle*, int *flags*, const char **fmtpath*, ...);

int **maapi_save_config_result**(int *sock*, int *id*);

int **maapi_load_config**(int *sock*, int *thandle*, int *flags*, const char **filename*);

int **maapi_load_config_cmds**(int *sock*, int *thandle*, int *flags*, const char **cmds*, const char **fmt*, ...);

int **maapi_load_config_stream**(int *sock*, int *thandle*, int *flags*);

int **maapi_load_config_stream_result**(int *sock*, int *id*);

int **maapi_roll_config**(int *sock*, int *thandle*, const char **fmtpath*, ...);

int **maapi_roll_config_result**(int *sock*, int *id*);

int **maapi_get_stream_progress**(int *sock*, int *id*);

int **maapi_xpath_eval**(int *sock*, int *thandle*, const char **expr*, int (**result*)(confd_hkeypath_t **kp*, confd_value_t **v*, void **state*), void (**trace*)(char **), void **initstate*, const char **fmtpath*, ...);

int **maapi_xpath_eval_expr**(int *sock*, int *thandle*, const char **expr*, char ***res*, void (**trace*)(char **), const char **fmtpath*, ...);

int **maapi_query_start**(int *sock*, int *thandle*, const char **expr*, const char **context_node*, int *chunk_size*, int *initial_offset*, enum confd_query_result_type *result_as*, int *nselect*, const char **select[]*, int *nsort*, const char **sort[]*);

int **maapi_query_startv**(int *sock*, int *thandle*, const char **expr*, const char **context_node*, int *chunk_size*, int *initial_offset*, enum confd_query_result_type *result_as*, int *select_nparams*, ...);

int **maapi_query_result**(int *sock*, int *qh*, struct confd_query_result ***qrs*);

int **maapi_query_result_count**(int *sock*, int *qh*);

int **maapi_query_free_result**(struct confd_query_result **qrs*);

```
int maapi_query_reset_to(int sock, int qh, int offset);

int maapi_query_reset(int sock, int qh);

int maapi_query_stop(int sock, int qh);

int maapi_do_display(int sock, int thandle, const char *fmtpath, ...);

int maapi_install_crypto_keys(int sock);

int maapi_init_upgrade(int sock, int timeoutsecs, int flags);

int maapi_perform_upgrade(int sock, const char **loadpathdirs, int n);

int maapi_commit_upgrade(int sock);

int maapi_abort_upgrade(int sock);

int maapi_aaa_reload(int sock, int synchronous);

int maapi_aaa_reload_path(int sock, int synchronous, const char
*fmt, ...);

int maapi_start_phase(int sock, int phase, int synchronous);

int maapi_wait_start(int sock, int phase);

int maapi_reload_config(int sock);

int maapi_reopen_logs(int sock);

int maapi_stop(int sock, int synchronous);

int maapi_rebind_listener(int sock, int listener);

int maapi_clear_opcache(int sock, const char *fmt, ...);
```

LIBRARY

ConfD Library, (`libconfd`, `-lconfd`)

DESCRIPTION

The `libconfd` shared library is used to connect to the ConfD transaction manager. The API described in this man page has several purposes. We can use MAAPI when we wish to implement our own proprietary management agent. We also use MAAPI to attach to already existing ConfD transactions, for example when we wish to implement semantic validation of configuration data in C, and also when we wish to implement CLI wizards in C.

PATHS

The majority of the functions described here take as their two last arguments a format string and a variable number of extra arguments as in: char * *`fmt`*, `...`);

The paths for MAAPI work like paths for CDB (see confd_lib_cdb(3)) with the exception that the bracket notation '[n]' is not allowed for MAAPI paths.

All the functions that take a path on this form also have a va_list variant, of the same form as `maapi_vget_elem()` and `maapi_vset_elem()`, which are the only ones explicitly documented below. I.e. they have a prefix "maapi_v" instead of "maapi_", and take a single va_list argument instead of a variable number of arguments.

FUNCTIONS

All functions return CONFD_OK (0), CONFD_ERR (-1) or CONFD_EOF (-2) unless otherwise stated. Whenever CONFD_ERR is returned from any API function in confd_lib_maapi it is possible to obtain additional information on the error through the symbol `confd_errno`, see the ERRORS section of confd_lib_lib(3).

In the case of CONFD_EOF it means that the socket to ConfD has been closed.

`int` **`maapi_connect`**`(int` *`sock`*`, const struct sockaddr*` *`srv`*`, int` *`srv_sz`*`);`

The application has to connect to ConfD before it can interact with ConfD .

Note

If this call fails (i.e. does not return CONFD_OK), the socket descriptor must be closed and a new socket created before the call is re-attempted.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_load_schemas`**`(int` *`sock`*`);`

This function dynamically loads schema information from the ConfD daemon into the library, where it is available to all the library components as described in the confd_types(3) and confd_lib_lib(3) man pages. See also `confd_load_schemas()` in confd_lib_lib(3).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_load_schemas_list`**`(int` *`sock`*`, int` *`flags`*`, const u_int32_t *`*`nshash`*`, const int *`*`nsflags`*`, int` *`num_ns`*`);`

A variant of `maapi_load_schemas()` that allows for loading a subset of the schema information from the ConfD daemon into the library. This means that the loading can be significantly faster in the case of a system with many large data models, with the drawback that the functions that use the schema information will have limited functionality or not work at all.

The *`flags`* parameter can be given as `CONFD_LOAD_SCHEMA_HASH` to request that the global mapping between strings and hash values for the data model nodes should be loaded. If *`flags`* is given as 0, this mapping is not loaded. The mapping is required for use of the functions `confd_hash2str()`, `confd_str2hash()`, `confd_cs_node_cd()`, and `confd_xpath_pp_kpath()`. Additionally, without the mapping, `confd_pp_value()`, `confd_pp_kpath()`, and `confd_pp_kpath_len()`, as well as the trace printouts from the library, will print nodes as "tag<N>", where N is the hash value, instead of the node name.

The *`nshash`* parameter is a *`num_ns`* elements long array of namespace hash values, requesting that schema information should be loaded for the listed namespaces according to the corresponding element of the *`nsflags`* array (also *`num_ns`* elements long). For each namespace, either or both of these flags may be given:

`CONFD_LOAD_SCHEMA_NODES` This flag requests that the confd_cs_node tree (see confd_types(3)) for the namespace should be loaded.

This tree is required for the use of the functions `confd_find_cs_root()`, `confd_find_cs_node()`, `confd_find_cs_node_child()`, `confd_cs_node_cd()`, `confd_register_node_type()`, `confd_get_leaf_list_type()`, and `confd_xpath_pp_kpath()` for the namespace. Additionally, the above functions that print a confd_hkeypath_t, as well as the library trace printouts, will attempt to use this tree and the type information (see below) to find the correct string representation for key values - if the tree isn't available, key values will be printed as described for `confd_pp_value()`.

`CONFD_LOAD_SCHEMA_TYPES` This flag requests that information about the types defined in the namespace should be loaded. The type information is required for use of the functions `confd_val2str()`, `confd_str2val()`, `confd_find_ns_type()`, `confd_get_leaf_list_type()`, `confd_register_ns_type()`, and `confd_register_node_type()` for the namespace. Additionally the confd_hkeypath_t-printing functions and the library trace printouts will also fall back to `confd_pp_value()` as described above if the type information isn't available.

Type definitions may refer to types defined in other namespaces. If the `CONFD_LOAD_SCHEMA_TYPES` flag has been given for a namespace, and the types defined there have such type references to namespaces that are not included in the *`nshash`* array, the referenced type information will also be loaded, if necessary recursively, until the types have a complete definition.

See also `confd_load_schemas_list()` in confd_lib_lib(3).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

```
int maapi_close(int sock);
```

Effectively a call to `maapi_end_user_session()` and also closes the socket.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION

Even if the call returns an error, the socket will be closed.

SESSION MANAGEMENT

```
int maapi_start_user_session(int sock, const char *username, const char
*context, const char **groups, int numgroups, const struct confd_ip
*src_addr, enum confd_proto prot);
```

Once we have created a MAAPI socket, we must also establish a user session on the socket. It is up to the user of the MAAPI library to authenticate users. The library user can ask ConfD to perform the actual authentication through a call to `maapi_authenticate()` but authentication may very well occur through some other external means.

Thus, when we use this function to create a user session, we must provide all relevant information about the user. If we wish to execute read/write transactions over the MAAPI interface, we must first have an established user session.

A user session corresponds to a NETCONF manager who has just established an authenticated SSH connection, but not yet sent any NETCONF commands on the SSH connection.

The struct confd_ip is defined in `confd_lib.h` and must be properly populated before the call. For example:

```
struct confd_ip ip;
ip.af = AF_INET;
inet_aton("10.0.0.33", &ip.ip.v4);
```

The *`context`* parameter can be any string. The string provided here is precisely the context string which will be used to authorize all data access through the AAA system. Each AAA rule has a context string which must match in order for a AAA rule to match. (See the AAA chapter in the User Guide.)

Using the string "system" for *`context`* has special significance:

- The session is exempt from all `maxSessions` limits in confd.conf.
- There will be no authorization checks done by the AAA system.
- The session is not logged in the audit log.
- The session is not shown in 'confd --status', nor 'show users' in CLI etc.
- The session may be started already in ConfD start phase 0. (However read-write transactions can not be started until phase 1, i.e. transactions started in phase 0 must use parameter *`readwrite`* == `CONFD_READ`).

Thus this can be useful e.g. when we need to create the user session for an "internal" transaction done by an application, without relation to a session from a northbound agent. Of course the implications of the above need to be carefully considered in each case.

It is not possible to create new user sessions until ConfD has reached start phase 2 (See confd(1)), with the above exception of a session with the context set to "system".

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_ALREADY_EXISTS, CONFD_ERR_BADSTATE

`int` **`maapi_start_user_session2`**`(int` *`sock`*`, const char *`*`username`*`, const char *`*`context`*`, const char **`*`groups`*`, int` *`numgroups`*`, const struct confd_ip *`*`src_addr`*`, int` *`src_port`*`, enum confd_proto` *`prot`*`);`

This function does the same as `maapi_start_user_session()`, but allows for the TCP/UDP source port to be passed to ConfD . Calling `maapi_start_user_session()` is equivalent to calling `maapi_start_user_session2()` with *`src_port`* 0.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_ALREADY_EXISTS, CONFD_ERR_BADSTATE

`int` **`maapi_end_user_session`**`(int` *`sock`*`);`

Ends our own user session. If the MAAPI socket is closed, the user session is automatically ended.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION

`int` **`maapi_get_my_user_session_id`**`(int` *`sock`*`);`

A user session is identified through an integer index, a usessid. This function returns the usessid associated with the MAAPI socket *sock*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_get_user_sessions`**`(int` *`sock`*`, int` *`res[]`*`, int` *`n`*`);`

Get the usessid for all current user sessions. The *res* array is populated with at most *n* usessids, and the total number of user sessions is returned (i.e. if the return value is larger than *n*, the array was too short to hold all usessids).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_kill_user_session`**`(int` *`sock`*`, int` *`usessid`*`);`

Kill the user session identified by *usessid*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_get_user_session`**`(int` *`sock`*`, int` *`usessid`*`, struct confd_user_info *`*`us`*`);`

Populate the confd_user_info structure with the data for the user session identified by *usessid*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_get_user_session_identification`**`(int` *`sock`*`, int` *`usessid`*`, struct confd_user_identification *`*`uident`*`);`

If the flag `CONFD_USESS_FLAG_HAS_IDENTIFICATION` is set in the `flags` field of the confd_user_info structure, additional identification information has been provided by the northbound client. This information can then be retrieved into a confd_user_identification structure (see `confd_lib.h`) by calling this function. The elements of confd_user_identification are either NULL (if the corresponding information was not provided) or point to a string. The strings must be freed by the application by means of calling `free(3)`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_get_user_session_opaque`**`(int` *`sock`*`, int` *`usessid`*`, char **`*`opaque`*`);`

If the flag `CONFD_USESS_FLAG_HAS_OPAQUE` is set in the `flags` field of the confd_user_info structure, "opaque" information has been provided by the northbound client (see the `-O` option in confd_cli(1)). The information can then be retrieved by calling this function. If the call is successful, *opaque* is set to point to a dynamically allocated string, which must be freed by the application by means of calling `free(3)`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_get_authorization_info`**`(int` *`sock`*`, int` *`usessid`*`, struct confd_authorization_info **`*`ainfo`*`);`

This function retrieves authorization info for a user session, i.e. the groups that the user has been assigned to. The struct confd_authorization_info is defined as:

```
struct confd_authorization_info {
    int ngroups;
```

```
    char **groups;
};
```

If the call is successful, *ainfo* is set to point to a dynamically allocated structure, which must be freed by the application by means of calling `confd_free_authorization_info()` (see confd_lib_lib(3)) .

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_set_user_session`**`(int` *`sock`*`, int` *`usessid`*`);`

Associate the socket with an already existing user session. This can be used instead of `maapi_start_user_session()` when we really do not want to start a new user session, e.g. if we want to call an action on behalf of a given user session.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

LOCKS

`int` **`maapi_lock`**`(int` *`sock`*`, enum confd_dbname` *`name`*`);`

`int` **`maapi_unlock`**`(int` *`sock`*`, enum confd_dbname` *`name`*`);`

These functions can be used to manipulate locks on the 3 different database types. If `maapi_lock()` is called and the database is already locked, CONFD_ERR is returned, and `confd_errno` will be set to CONFD_ERR_LOCKED. If `confd_errno` is CONFD_ERR_EXTERNAL it means that a callback has been invoked in an external database to lock/unlock which in its turn returned an error. (See confd_lib_dp(3) for external database callback API)

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_LOCKED, CONFD_ERR_EXTERNAL, CONFD_ERR_NOSESSION

`int` **`maapi_is_lock_set`**`(int` *`sock`*`, enum confd_dbname` *`name`*`);`

Returns a positive integer being the usid of the current lock owner if the lock is set, and 0 if the lock is not set.

`int` **`maapi_lock_partial`**`(int` *`sock`*`, enum confd_dbname` *`name`*`, char *`*`xpaths[]`*`, int` *`nxpaths`*`, int *`*`lockid`*`);`

`int` **`maapi_unlock_partial`**`(int` *`sock`*`, int` *`lockid`*`);`

We can also manipulate partial locks on the databases, i.e. locks on a specified set of leafs and/or subtrees. The specification of what to lock is given via the *`xpaths`* array, which is populated with *`nxpaths`* pointers to XPath expressions. If the lock succeeds, `maapi_lock_partial()` returns CONFD_OK, and a lock identifier to use with `maapi_unlock_partial()` is stored in *`*lockid`*.

If CONFD_ERR is returned, some values of `confd_errno` are of particular interest:

CONFD_ERR_LOCKED Some of the requested nodes are already locked.

CONFD_ERR_EXTERNAL A callback has been invoked in an external database to lock_partial/unlock_partial which in its turn returned an error (see confd_lib_dp(3) for external database callback API).

CONFD_ERR_NOEXISTS The list of XPath expressions evaluated to an empty set of nodes - i.e. there is nothing to lock.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_LOCKED, CONFD_ERR_EXTERNAL, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

CANDIDATE MANIPULATION

All the candidate manipulation functions require that the candidate data store is enabled in `confd.conf` - otherwise they will set `confd_errno` to CONFD_ERR_NOEXISTS. If the candidate data store is enabled, `confd_errno` may be set to CONFD_ERR_NOEXISTS for other reasons, as described below.

All these functions may also set `confd_errno` to CONFD_ERR_EXTERNAL. This value can only be set when the candidate is owned by the external database. When ConfD owns the candidate, which is the most common configuration scenario, the candidate manipulation function will never set `confd_errno` to CONFD_ERR_EXTERNAL.

`int` **`maapi_candidate_validate`**`(int` *`sock`*`);`

This function validates the candidate. The function should only be used when the candidate is not owned by ConfD , i.e. when the candidate is owned by an external database.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_commit`**`(int` *`sock`*`);`

This function copies the candidate to running. It is also used to confirm a previous call to `maapi_candidate_confirmed_commit()`, i.e. to prevent the automatic rollback if a confirmed commit is not confirmed.

If `confd_errno` is CONFD_ERR_INUSE it means that some other user session is doing a confirmed commit or has a lock on the database. CONFD_ERR_NOEXISTS means that there is an ongoing persistent confirmed commit (see below) - i.e. there is no confirmed commit that this function call can apply to.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_INUSE, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_confirmed_commit`**`(int` *`sock`*`, int` *`timeoutsecs`*`);`

This function also copies the candidate into running. However if a call to `maapi_candidate_commit()` is not done within *`timeoutsecs`* an automatic rollback will occur. It can also be used to "extend" a confirmed commit that is already in progress, i.e. set a new timeout or add changes.

If `confd_errno` is CONFD_ERR_NOEXISTS it means that there is an ongoing persistent confirmed commit (see below).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_INUSE, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_abort_commit`**`(int` *`sock`*`);`

This function cancels an ongoing confirmed commit.

If `confd_errno` is CONFD_ERR_NOEXISTS it means that some other user session initiated the confirmed commit, or that there is an ongoing persistent confirmed commit (see below).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_confirmed_commit_persistent`**`(int sock, int timeout-secs, const char *persist, const char *persist_id);`

This function can be used to start or extend a persistent confirmed commit. The *`persist`* parameter sets the cookie for the persistent confirmed commit, while the *`persist_id`* gives the cookie for an already ongoing persistent confirmed commit. This gives the following possibilities:

`persist` = "cookie", *`persist_id`* = NULL	Start a persistent confirmed commit with the cookie "cookie", or extend an already ongoing non-persistent confirmed commit and turn it into a persistent confirmed commit.
`persist` = "newcookie", *`persist_id`* = "oldcookie"	Extend an ongoing persistent confirmed commit that uses the cookie "oldcookie" and change the cookie to "newcookie".
`persist` = NULL, *`persist_id`* = "cookie"	Extend an ongoing persistent confirmed commit that uses the cookie "oldcookie" and turn it into a non-persistent confirmed commit.
`persist` = NULL, *`persist_id`* = NULL	Does the same as `maapi_candidate_commit_confirmed()`.

Typical usage is to start a persistent confirmed commit with `persist` = "cookie", `persist_id` = NULL, and to extend it with `persist` = "cookie", `persist_id` = "cookie".

If `confd_errno` is CONFD_ERR_NOEXISTS it means that there is an ongoing persistent confirmed commit, but `persist_id` didn't give the right cookie for it.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_INUSE, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_commit_persistent`**`(int sock, const char *persist_id);`

Confirm an ongoing persistent commit with the cookie given by `persist_id`. (If `persist_id` is NULL, it does the same as `maapi_candidate_commit()`).

If `confd_errno` is CONFD_ERR_NOEXISTS it means that there is an ongoing persistent confirmed commit, but `persist_id` didn't give the right cookie for it.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_INUSE, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_abort_commit_persistent`**`(int sock, const char *persist_id);`

Cancel an ongoing persistent commit with the cookie given by `persist_id`. (If `persist_id` is NULL, it does the same as `maapi_candidate_abort_commit()`.)

If `confd_errno` is CONFD_ERR_NOEXISTS it means that there is an ongoing persistent confirmed commit, but `persist_id` didn't give the right cookie for it.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_INUSE, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_candidate_reset`**`(int sock);`

This function copies running into candidate.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_INUSE, CONFD_ERR_EXTERNAL, CONFD_ERR_NOSESSION

`int` **`maapi_confirmed_commit_in_progress`**`(int` *`sock`*`);`

Checks whether a confirmed commit is ongoing. Returns 1 if some user session currently has an ongoing confirmed commit operation and 0 if not.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_copy_running_to_startup`**`(int` *`sock`*`);`

This function copies running to startup.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_INUSE, CONFD_ERR_EXTERNAL, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

`int` **`maapi_is_running_modified`**`(int` *`sock`*`);`

Returns 1 if running has been modified since the last copy to startup, 0 if it has not been modified.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

`int` **`maapi_is_candidate_modified`**`(int` *`sock`*`);`

Returns 1 if candidate has been modified, i.e if there are any outstanding non committed changes to the candidate, 0 if no changes are done

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

TRANSACTION CONTROL

`int` **`maapi_start_trans`**`(int` *`sock`*`, enum confd_dbname` *`name`*`, enum confd_trans_mode` *`readwrite`*`);`

The main purpose of MAAPI is to provide read and write access into the ConfD transaction manager. Regardless of whether data is kept in CDB or in some (or several) external data bases, the same API is used to access data. ConfD acts as a mediator and multiplexes the different commands to the code which is responsible for each individual data node.

This function creates a new transaction towards the data store specified by *`name`*, which can be one of `CONFD_CANDIDATE`, `CONFD_RUNNING`, or `CONFD_STARTUP` (however updating the startup data store is better done via `maapi_copy_running_to_startup()`). The *`readwrite`* parameter can be either `CONFD_READ`, to start a readonly transaction, or `CONFD_READ_WRITE`, to start a read-write transaction.

A readonly transaction will incur less resource usage, thus if no writes will be done (e.g. the purpose of the transaction is only to read operational data), it is best to use `CONFD_READ`. There are also some cases where starting a read-write transaction is not allowed, e.g. if we start a transaction towards the running data store and `/confdConfig/datastores/running/access` is set to "writable-through-candidate" in `confd.conf`, or if ConfD is running in HA slave mode.

If start of the transaction is successful, the function returns a new transaction handle, a non-negative integer `thandle` which must be used as a parameter in all API functions which manipulate the transaction.

We will drive this transaction forward through the different states a ConfD transaction goes through. See the ascii arts in confd_lib_dp(3) for a picture of these states. If an external database is used, and it has registered callback functions for the different transaction states, those callbacks will be called when we in MAAPI invoke the different MAAPI transaction manipulation functions. For example when we call `maapi_start_trans()` the `init()` callback will be invoked in all external databases. (However ConfD may delay the actual invocation of `init()` as an optimization, see confd_lib_dp(3).) If data is kept in CDB, ConfD will handle everything internally.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_TOOMANYTRANS, CONFD_ERR_BADSTATE, CONFD_ERR_NOT_WRITABLE

`int` **`maapi_start_trans2`**`(int` *`sock`*`, enum confd_dbname` *`name`*`, enum confd_trans_mode` *`readwrite`*`, int` *`usid`*`);`

If we want to start new transactions inside actions, we can use this function to execute the new transaction within the existing user session. It is equivalent to calling `maapi_set_user_session()` and then `maapi_start_trans()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_TOOMANYTRANS, CONFD_ERR_BADSTATE, CONFD_ERR_NOT_WRITABLE

`int` **`maapi_start_trans_flags`**`(int` *`sock`*`, enum confd_dbname` *`name`*`, enum confd_trans_mode` *`readwrite`*`, int` *`usid`*`, int` *`flags`*`);`

This function makes it possible to set the flags that can otherwise be used with `maapi_set_flags()` already when starting a transaction, as well as setting the `MAAPI_FLAG_HIDE_INACTIVE` and `MAAPI_FLAG_DELAYED_WHEN` flags that can only be used with `maapi_start_trans_flags()`. See the description of `maapi_set_flags()` for the available flags. It also incorporates the functionality of `maapi_start_trans()` and `maapi_start_trans2()` with respect to user sessions: If *`usid`* is 0, the transaction will be started within the user session associated with the MAAPI socket (like `maapi_start_trans()`), otherwise it will be started within the user session given by *`usid`* (like `maapi_start_trans2()`).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_TOOMANYTRANS, CONFD_ERR_BADSTATE, CONFD_ERR_NOT_WRITABLE

`int` **`maapi_start_trans_in_trans`**`(int` *`sock`*`, enum confd_trans_mode` *`readwrite`*`, int` *`usid`*`, int` *`thandle`*`);`

This function makes it possible to start a transaction with another transaction as backend, instead of an actual data store. This can be useful if we want to make a set of related changes, and then either apply or discard them all based on some criterion, while other changes remain unaffected. The *`thandle`* identifies the backend transaction to use. If *`usid`* is 0, the transaction will be started within the user session associated with the MAAPI socket, otherwise it will be started within the user session given by *`usid`*. If we call `maapi_apply_trans()` for this "transaction in a transaction", the changes (if any) will be applied to the backend transaction. To discard the changes, call `maapi_finish_trans()` without calling `maapi_apply_trans()` first.

The changes in this transaction can be validated by calling `maapi_validate_trans()` with a non-zero value for *`forcevalidation`*, but calling `maapi_apply_trans()` will not do any validation - in either case, the resulting configuration will be validated when the backend transaction is committed to the running data store. Note though that unlike the case with a transaction directly towards a data store, no transaction lock is taken on the underlying data store when doing validation of this type of transaction - thus it is possible for the contents of the data store to change (due to commit of another transaction) during the validation.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_TOOMANYTRANS, CONFD_ERR_BADSTATE

`int` **`maapi_finish_trans`**`(int` *`sock`*`, int` *`thandle`*`);`

This will finish the transaction. If the transaction is implemented by an external database, this will invoke the `finish()` callback.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

The error CONFD_ERR_NOEXISTS is set for all API functions which use a *`thandle`*, the return value from `maapi_start_trans()`, whenever no transaction is started.

`int` **`maapi_validate_trans`**`(int` *`sock`*`, int` *`thandle`*`, int` *`unlock`*`, int` *`forcevalidation`*`);`

This function validates all data written in the transaction. This includes all data model constraints and all defined semantic validation in C, i.e. user programs that have registered functions under validation points. (See the Semantic Validation chapter in the User Guide.)

If this function returns CONFD_ERR, the transaction is open for further editing. There are two special `confd_errno` values which are of particular interest here.

CONFD_ERR_EXTERNAL
: this means that an external validation program in C returns CONFD_ERR i.e. that the semantic validation failed. The reason for the failure can be found in `confd_lasterr()`

CONFD_ERR_VALIDATION_WARNING
: this means that an external semantic validation program in C returned CONFD_VALIDATION_WARN. The string `confd_lasterr()` is organized as a series of NUL terminated strings as in `keypath1, reason1, keypath2, reason2 ...` where the sequence is terminated with an additional NUL

If *`unlock`* is 1, the transaction is open for further editing even if validation succeeds. If *`unlock`* is 0 and the function returns CONFD_OK, the next function to be called MUST be `maapi_prepare_trans()` or `maapi_finish_trans()`.

`unlock` = 1 can be used to implement a 'validate' command which can be given in the middle of an editing session. The first thing that happens is that a lock is set. If *`unlock`* == 1, the lock is released on success. The lock is always released on failure.

The *`forcevalidation`* parameter should normally be 0. It has no effect for a transaction towards the running or startup data stores, validation is always performed. For a transaction towards the candidate data store, validation will not be done unless *`forcevalidation`* is non-zero. Avoiding this validation is preferable if we are going to commit the candidate to running (e.g. with `maapi_candidate_commit()`), since otherwise the validation will be done twice. However if we are implementing a 'validate' command, we should give a non-zero value for *`forcevalidation`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTSET, CONFD_ERR_NON_UNIQUE, CONFD_ERR_BAD_KEYREF, CONFD_ERR_TOO_FEW_ELEMS, CONFD_ERR_TOO_MANY_ELEMS, CONFD_ERR_UNSET_CHOICE, CONFD_ERR_MUST_FAILED, CONFD_ERR_MISSING_INSTANCE, CONFD_ERR_INVALID_INSTANCE, CONFD_ERR_INUSE, CONFD_ERR_BADTYPE, CONFD_ERR_EXTERNAL, CONFD_ERR_BADSTATE

```
int maapi_prepare_trans(int sock, int thandle);
```

This function must be called as first part of two-phase commit. After this function has been called `maapi_commit_trans()` or `maapi_abort_trans()` must be called.

It will invoke the prepare callback in all participants in the transaction. If all participants reply with CONFD_OK, the second phase of the two-phase commit procedure is commenced.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS, CONFD_ERR_EXTERNAL, CONFD_ERR_NOTSET, CONFD_ERR_BADSTATE, CONFD_ERR_INUSE

```
int maapi_commit_trans(int sock, int thandle);
```

```
int maapi_abort_trans(int sock, int thandle);
```

Finally at the last stage, either commit or abort must be called. A call to one of these functions must also eventually be followed by a call to `maapi_finish_trans()` which will terminate the transaction.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS, CONFD_ERR_EXTERNAL, CONFD_ERR_BADSTATE

```
int maapi_apply_trans(int sock, int thandle, int keepopen);
```

Invoking the above transaction functions in exactly the right order can be a bit complicated. The right order to invoke the functions is `maapi_validate_trans()`, `maapi_prepare_trans()`, `maapi_commit_trans()` (or `maapi_abort_trans()`). Usually we do not require this fine grained control over the two-phase commit protocol. It is easier to use `maapi_apply_trans()` which validates, prepares and eventually commits or aborts.

A call to `maapi_apply_trans()` must also eventually be followed by a call to `maapi_finish_trans()` which will terminate the transaction.

Note

For a readonly transaction, i.e. one started with *`readwrite`* == `CONFD_READ`, or for a read-write transaction where we haven't actually done any writes, we do not need to call any of the validate/prepare/commit/abort or apply functions, since there is nothing for them to do. Calling `maapi_finish_trans()` to terminate the transaction is sufficient.

The parameter *`keepopen`* can optionally be set to `1`, then the changes to the transaction are not discarded if validation fails. This feature is typically used by management applications that wish to present the validation errors to an operator and allow the operator to fix the validation errors and then later retry the apply sequence.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTSET, CONFD_ERR_NON_UNIQUE, CONFD_ERR_BAD_KEYREF, CONFD_ERR_TOO_FEW_ELEMS, CONFD_ERR_TOO_MANY_ELEMS, CONFD_ERR_UNSET_CHOICE, CONFD_ERR_MUST_FAILED, CONFD_ERR_MISSING_INSTANCE, CONFD_ERR_INVALID_INSTANCE, CONFD_ERR_INUSE, CONFD_ERR_BADTYPE, CONFD_ERR_EXTERNAL, CONFD_ERR_BADSTATE

READ/WRITE FUNCTIONS

```
int maapi_set_namespace(int sock, int thandle, int hashed_ns);
```

If we want to read or write data where the toplevel element name is not unique, we must indicate which namespace we are going to use. It is possible to change the namespace several times during a transaction.

The *hashed_ns* integer is the integer which is defined for the namespace in the .h file which is generated by the 'confdc' compiler. It is also possible to indicate which namespace to use through the namespace prefix when we read and write data. Thus the path `/foo:bar/baz` will get us `/bar/baz` in the namespace with prefix "foo" regardless of what the "set" namespace is. And if there is only one toplevel element called "bar" across all namespaces, we can use `/bar/baz` without the prefix and without calling `maapi_set_namespace()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

```
int maapi_cd(int sock, int thandle, const char *fmt, ...);
```

This function mimics the behavior of the UNIX "cd" command. It changes our working position in the data tree. If we are worried about performance, it is more efficient to invoke `maapi_cd()` to some position in the tree and there perform a series of operations using relative paths than it is to perform the equivalent series of operations using absolute paths. Note that this function can not be used as an existence test.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS

```
int maapi_pushd(int sock, int thandle, const char *fmt, ...);
```

Behaves like `maapi_cd()` with the exception that we can subsequently call `maapi_popd()` and returns to the previous position in the data tree.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOSTACK, CONFD_ERR_NOEXISTS

```
int maapi_popd(int sock, int thandle);
```

Pops the top position of the directory stack and changes directory.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOSTACK, CONFD_ERR_NOEXISTS

```
int maapi_getcwd(int sock, int thandle, size_t strsz, char *curdir);
```

Returns the current position as previously set by `maapi_cd()`, `maapi_pushd()`, or `maapi_popd()` as a string. Note that what is returned is a pretty-printed version of the internal representation of the current position, it will be the shortest unique way to print the path but it might not exactly match the string given to `maapi_cd()`. The buffer in *curdir will be NULL terminated, and no more characters than strsz-1 will be written to it.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

```
int maapi_getcwd_kpath(int sock, int thandle, confd_hkeypath_t **kp);
```

Returns the current position like `maapi_getcwd()`, but as a pointer to a hashed keypath instead of as a string. The hkeypath is dynamically allocated, and may further contain dynamically allocated elements. The caller must free the allocated memory, easiest done by calling `confd_free_hkeypath()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

`int` **`maapi_exists`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`fmt`*`, ...);`

Boolean function which return 1 if the path refers to an existing node in the data tree, 0 if it does not.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_num_instances`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`fmt`*`, ...);`

Returns the number of entries for a list in the data tree.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_get_elem`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t *`*`v`*`, const char *`*`fmt`*`, ...);`

This reads a value from the path in *`fmt`* and writes the result into the result parameter confd_value_t. The path must lead to a leaf node in the data tree. Note that for the C_BUF, C_BINARY, C_LIST, C_OBJECTREF, C_OID, C_QNAME, and C_HEXSTR confd_value_t types the buffer(s) pointed to are allocated using malloc(3), it is up to the user of this interface to free them using `confd_free_value()`.

The maapi interface also contains a long list of access functions that accompany the `maapi_get_elem()` function which is a general access function that returns a confd_value_t. The accompanying functions all have the format `maapi_get_<type>_elem()` where <type> is one of the actual C types a confd_value_t can have. For example the function:

```
maapi_get_int64_elem(int sock, int thandle, int64_t *rval,
                                 const char *fmt, ...);
```

is used to read a signed 64 bit integer. It fills in the provided int64_t parameter. This corresponds to the YANG datatype int64, see confd_types(3). Similar access functions are provided for all the different builtin types.

One access function that needs additional explaining is the `maapi_get_str_elem()`. This function copies at most `n-1` characters into a user provided buffer, and terminates the string with a NUL character. If the buffer is not sufficiently large CONFD_ERR is returned, and `confd_errno` is set to CONFD_ERR_PROTOUSAGE. Note it is always possible to use maapi_get_elem() to get hold of the confd_value_t, which in the case of a string buffer contains the length.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_PROTOUSAGE, CONFD_ERR_BADTYPE

`int` **`maapi_get_int8_elem`**`(int` *`sock`*`, int` *`thandle`*`, int8_t *`*`rval`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_get_int16_elem`**`(int` *`sock`*`, int` *`thandle`*`, int16_t *`*`rval`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_get_int32_elem`**`(int` *`sock`*`, int` *`thandle`*`, int32_t *`*`rval`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_get_int64_elem`**`(int` *`sock`*`, int` *`thandle`*`, int64_t *`*`rval`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_get_u_int8_elem`**`(int` *`sock`*`, int` *`thandle`*`, u_int8_t *`*`rval`*`, const char *`*`fmt`*`, ...);`

int **maapi_get_u_int16_elem**(int *sock*, int *thandle*, u_int16_t *rval, const char *fmt, ...);

int **maapi_get_u_int32_elem**(int *sock*, int *thandle*, u_int32_t *rval, const char *fmt, ...);

int **maapi_get_u_int64_elem**(int *sock*, int *thandle*, u_int64_t *rval, const char *fmt, ...);

int **maapi_get_ipv4_elem**(int *sock*, int *thandle*, struct in_addr *rval, const char *fmt, ...);

int **maapi_get_ipv6_elem**(int *sock*, int *thandle*, struct in6_addr *rval, const char *fmt, ...);

int **maapi_get_double_elem**(int *sock*, int *thandle*, double *rval, const char *fmt, ...);

int **maapi_get_bool_elem**(int *sock*, int *thandle*, int *rval, const char *fmt, ...);

int **maapi_get_datetime_elem**(int *sock*, int *thandle*, struct confd_datetime *rval, const char *fmt, ...);

int **maapi_get_date_elem**(int *sock*, int *thandle*, struct confd_date *rval, const char *fmt, ...);

int **maapi_get_gyearmonth_elem**(int *sock*, int *thandle*, struct confd_gYearMonth *rval, const char *fmt, ...);

int **maapi_get_gyear_elem**(int *sock*, int *thandle*, struct confd_gYear *rval, const char *fmt, ...);

int **maapi_get_time_elem**(int *sock*, int *thandle*, struct confd_time *rval, const char *fmt, ...);

int **maapi_get_gday_elem**(int *sock*, int *thandle*, struct confd_gDay *rval, const char *fmt, ...);

int **maapi_get_gmonthday_elem**(int *sock*, int *thandle*, struct confd_gMonthDay *rval, const char *fmt, ...);

int **maapi_get_month_elem**(int *sock*, int *thandle*, struct confd_gMonth *rval, const char *fmt, ...);

int **maapi_get_duration_elem**(int *sock*, int *thandle*, struct confd_duration *rval, const char *fmt, ...);

int **maapi_get_enum_value_elem**(int *sock*, int *thandle*, int32_t *rval, const char *fmt, ...);

int **maapi_get_bit32_elem**(int *sock*, int *th*, int32_t *rval, const char *fmt, ...);

int **maapi_get_bit64_elem**(int *sock*, int *th*, int64_t *rval, const char *fmt, ...);

```
int maapi_get_oid_elem(int sock, int th, struct confd_snmp_oid **rval,
const char *fmt, ...);

int maapi_get_buf_elem(int sock, int thandle, unsigned char **rval, int
*bufsiz, const char *fmt, ...);

int maapi_get_str_elem(int sock, int th, char *buf, int n, const char
*fmt, ...);

int maapi_get_binary_elem(int sock, int thandle, unsigned char **rval,
int *bufsiz, const char *fmt, ...);

int maapi_get_qname_elem(int sock, int thandle, unsigned char **prefix,
int *prefixsz, unsigned char **name, int *namesz, const char *fmt, ...);

int maapi_get_list_elem(int sock, int th, confd_value_t **values, int
*n, const char *fmt, ...);

int   maapi_get_ipv4prefix_elem(int   sock,   int   thandle,   struct
confd_ipv4_prefix *rval, const char *fmt, ...);

int   maapi_get_ipv6prefix_elem(int   sock,   int   thandle,   struct
confd_ipv6_prefix *rval, const char *fmt, ...);
```

Similar to the CDB API, MAAPI also includes typesafe variants for all the builtin types. See confd_types(3).

```
int maapi_vget_elem(int sock, int thandle, confd_value_t *v, const char
*fmt, va_list args);
```

This function does the same as `maapi_get_elem()`, but takes a single va_list argument instead of a variable number of arguments - i.e. similar to `vprintf()`. Corresponding va_list variants exist for all the functions that take a path as a variable number of arguments.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_PROTOUSAGE, CONFD_ERR_BADTYPE

```
int maapi_init_cursor(int sock, int thandle, struct maapi_cursor *mc,
const char *fmt, ...);
```

Whenever we wish to iterate over the entries in a list in the data tree, we must first initialize a cursor. The cursor is subsequently used in a while loop.

For example if we have:

```
container servers {
  list server {
    key name;
    max-elements 64;
    leaf name {
      type string;
    }
    leaf ip {
      type inet:ip-address;
    }
    leaf port {
      type inet:port-number;
```

```
      mandatory true;
    }
  }
}
```

We can have the following C code which iterates over all `server` entries.

```
struct maapi_cursor mc;

maapi_init_cursor(sock, th, &mc, "/servers/server");
maapi_get_next(&mc);
while (mc.n != 0) {
   ... do something
   maapi_get_next(&mc);
}
maapi_destroy_cursor(&mc);
```

When a `tailf:secondary-index` statement is used in the data model (see tailf_yang_extensions(5)), we can set the `secondary_index` element of the struct maapi_cursor to indicate the name of a chosen secondary index - this must be done after the call to `maapi_init_cursor()` (which sets `secondary_index` to NULL) and before any call to `maapi_get_next()`. In this case, `secondary_index` must point to a NUL-terminated string that is valid throughout the iteration.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_get_next`**`(struct maapi_cursor *`*`mc`*`);`

Iterates and gets the keys for the next entry in a list. The key(s) can be used to retrieve further data. The key(s) are stored as confd_value_t structures in an array inside the struct maapi_cursor. The array of keys will be deallocated by the library.

For example to read the `port` leaf from an entry in the `server` list above, we would do:

```
....
maapi_init_cursor(sock, th, &mc, "/servers/server");
maapi_get_next(&mc);
while (mc.n != 0) {
   confd_value_t v;
   maapi_get_elem(sock, th, &v, "/servers/server{%x}/port", &mc.keys[0]);
   ....
   maapi_get_next(&mc);
}
```

The '%*x' modifier (see the PATHS section in confd_lib_cdb(3)) is especially useful when working with a maapi cursor. The example above assumes that we know that the `/servers/server` list has exactly one key. But we can alternatively write `maapi_get_elem(sock, th, &v, "/servers/server{%*x}/port", mc.n, mc.keys);` - which works regardless of the number of keys that the list has.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_find_next`**`(struct maapi_cursor *`*`mc`*`, enum confd_find_next_type` *`type`*`, confd_value_t *`*`inkeys`*`, int` *`n_inkeys`*`);`

Update the cursor *`mc`* with the key(s) for the list entry designated by the *`type`* and *`inkeys`* parameters. This function may be used to start a traversal from an arbitrary entry in a list. Keys for subsequent entries may be retrieved with the `maapi_get_next()` function.

The *inkeys* array is populated with *n_inkeys* values that designate the starting point in the list. Normally the array is populated with key values for the list, but if the secondary_index element of the cursor has been set, the array must instead be populated with values for the corresponding secondary index-leafs. The *type* can have one of two values:

CONFD_FIND_NEXT — The keys for the first list entry *after* the one indicated by the *inkeys* array are requested. The *inkeys* array does not have to correspond to an actual existing list entry. Furthermore the number of values provided in the array (*n_inkeys*) may be fewer than the number of keys (or number of index-leafs for a secondary-index) in the data model, possibly even zero. This indicates that only the first *n_inkeys* values are provided, and the remaining ones should be taken to have a value "earlier" than the value for any existing list entry.

CONFD_FIND_SAME_OR_NEXT — If the values in the *inkeys* array completely identify an actual existing list entry, the keys for this entry are requested. Otherwise the same logic as described for CONFD_FIND_NEXT is used.

The following example will traverse the server list starting with the first entry (if any) that has a key value that is after "smtp" in the list order:

```
....
confd_value_t inkeys[1];

maapi_init_cursor(sock, th, &mc, "/servers/server");
CONFD_SET_STR(&inkeys[0], "smtp");

maapi_find_next(&mc, CONFD_FIND_NEXT, inkeys, 1);
while (mc.n != 0) {
   confd_value_t v;
   maapi_get_elem(sock, th, &v, "/servers/server{%x}/port", &mc.keys[0]);
   ....
   maapi_get_next(&mc);
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`void` **`maapi_destroy_cursor`**`(struct maapi_cursor *`*`mc`*`);`

Deallocates memory which is associated with the cursor.

`int` **`maapi_set_elem`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t *`*`v`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_set_elem2`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`strval`*`, const char *`*`fmt`*`, ...);`

We have two different functions to set values. One where the value is a string and one where the value to set is a confd_value_t. The string version is useful when we have implemented a management agent where the user enters values as strings. The version with confd_value_t is useful when we are setting values which we have just read.

Another note which might effect users is that if the type we are writing is any of the encrypt or hash types, the maapi_set_elem2() will perform the asymmetric conversion of values whereas the maapi_set_elem() will not. See confd_types(3), the types tailf:md5-digest-string, tailf:des3-cbc-encrypted-string, and tailf:aes-cfb-128-encrypted-string.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_INUSE

`int` **`maapi_vset_elem`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t *`*`v`*`, const char *`*`fmt`*`, va_list` *`args`*`);`

This function does the same as `maapi_set_elem()`, but takes a single va_list argument instead of a variable number of arguments - i.e. similar to `vprintf()`. Corresponding va_list variants exist for all the functions that take a path as a variable number of arguments.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_INUSE

`int` **`maapi_create`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`fmt`*`, ...);`

Create a new list entry, a `presence` container, or a leaf of type `empty` in the data tree. For example: `maapi_create(sock,th,"/servers/server{www}");`

If we are creating a new server entry as above, we must also populate all other data nodes below, which do not have a default value in the data model. Thus we must also do e.g.:

```
maapi_set_elem2(sock, th, "80", "/servers/server{www}/port");
```

before we try to commit the data.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOTCREATABLE, CONFD_ERR_INUSE, CONFD_ERR_ALREADY_EXISTS

`int` **`maapi_delete`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`fmt`*`, ...);`

Delete an existing list entry, a `presence` container, or an optional leaf and all its children (if any) from the data tree.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOTDELETABLE, CONFD_ERR_INUSE

`int` **`maapi_get_object`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t *`*`values`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

This function reads at most *`n`* values from the list entry or container specified by the path, and places them in the *`values`* array, which is provided by the caller. The array is populated according to the specification of the Value Array format in the XML STRUCTURES section of the confd_types(3) manual page.

On success, the function returns the actual number of elements needed. I.e. if the return value is bigger than *`n`*, only the values for the first *`n`* elements are in the array, and the remaining values have been discarded. Note that given the specification of the array contents, there is always a fixed upper bound on the number of actual elements, and if there are no `presence` sub-containers, the number is constant. See the description of `cdb_get_object()` in confd_lib_cdb(3) for usage examples - they apply to `maapi_get_object()` as well.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_get_objects`**`(struct maapi_cursor *`*`mc`*`, confd_value_t *`*`values`*`, int `*`n`*`, int *`*`nobj`*`);`

Similar to `maapi_get_object()`, but reads multiple list entries based on a struct maapi_cursor. At most *`n`* values from each of at most *`*nobj`* list entries, starting at the entry after the one given by *`*mc`*, are read and placed in the *`values`* array. The cursor must have been initialized with `maapi_init_cursor()` at some point before the call, but in principle it is possible to mix calls to `maapi_get_next()` and `maapi_get_objects()` using the same cursor.

The array must be at least `n * *nobj` elements long, and the values for entry `i` start at element `array[i * n]` (i.e. the first entry read starts at `array[0]`, the second at `array[n]`, and so on). On success, the highest actual number of values in any of the entries read is returned. If we attempt to read more entries than actually exist (i.e. if there are less than *`*nobj`* entries after the entry indicated by *`*mc`*), *`*nobj`* is updated with the actual number (possibly 0) of entries read. In this case the `n` element of the cursor is set to 0 as for `maapi_get_next()`. Example - read the data for all entries in the "server" list above, in chunks of 10:

```
#define VALUES_PER_ENTRY 3
#define ENTRIES_PER_REQUEST 10

struct maapi_cursor mc;
confd_value_t v[ENTRIES_PER_REQUEST*VALUES_PER_ENTRY];
int nobj, ret, i;

maapi_init_cursor(sock, th, &mc, "/servers/server");
do {
    nobj = ENTRIES_PER_REQUEST;
    ret = maapi_get_objects(&mc, v, VALUES_PER_ENTRY, &nobj);
    if (ret >= 0) {
        for (i = 0; i < nobj; i++) {
            ... process entry starting at v[i*VALUES_PER_ENTRY] ...
        }
    } else {
        ... handle error ...
    }
} while (ret >= 0 && mc.n != 0);
maapi_destroy_cursor(&mc);
```

See also the description of `cdb_get_object()` in confd_lib_cdb(3) for examples on how to use loaded schema information to avoid "hardwiring" constants like VALUES_PER_ENTRY above, and the relative position of individual leaf values in the value array.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_PROTOUSAGE, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_get_values`**`(int `*`sock`*`, int `*`thandle`*`, confd_tag_value_t *`*`values`*`, int `*`n`*`, const char *`*`fmt`*`, ...);`

Read an arbitrary set of sub-elements of a container or list entry. The *`values`* array must be pre-populated with *`n`* values based on the specification of the *Tagged Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page, where the confd_value_t value element is given as follows:

- C_NOEXISTS means that the value should be read from the transaction and stored in the array.
- C_PTR also means that the value should be read from the transaction, but instead gives the expected type and a pointer to the type-specific variable where the value should be stored. Thus this gives a functionality similar to the type safe `maapi_get_xxx_elem()` functions.

- C_XMLBEGIN and C_XMLEND are used as per the specification.
- Keys to select list entries can be given with their values.

Note

When we use C_PTR, we need to take special care to free any allocated memory. When we use C_NOEXISTS and the value is stored in the array, we can just use `confd_free_value()` regardless of the type, since the confd_value_t has the type information. But with C_PTR, only the actual value is stored in the pointed-to variable, just as for `maapi_get_buf_elem()`, `maapi_get_binary_elem()`, etc, and we need to free the memory specifically allocated for the types listed in the description of `maapi_get_elem()` above. The details of how to do this are not given for the `maapi_get_xxx_elem()` functions here, but it is the same as for the corresponding `cdb_get_xxx()` functions, see confd_lib_cdb(3).

All elements have the same position in the array after the call, in order to simplify extraction of the values - this means that optional elements that were requested but didn't exist will have C_NOEXISTS rather than being omitted from the array. However requesting a list entry that doesn't exist is an error. Note that when using C_PTR, the only indication of a non-existing value is that the destination variable has not been modified - it's up to the application to set it to some "impossible" value before the call when optional leafs are read.

Note

Selection of a list entry by its "instance integer", which can be done with `cdb_get_values()` by using C_CDBBEGIN, can *not* be done with `maapi_get_values()`

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

```
int maapi_set_object(int sock, int thandle, const confd_value_t *values,
int n, const char *fmt, ...);
```

Set all leafs corresponding to the complete contents of a list entry or container, excluding for sub-lists. The *`values`* array must be populated with *`n`* values according to the specification of the Value Array format in the XML STRUCTURES section of the confd_types(3) manual page. Additionally, since operational data cannot be written, array elements corresponding to operational data leafs or containers must have the value C_NOEXISTS.

If the node specified by the path, or any sub-nodes that are specified as existing, do not exist before this call, they will be created, otherwise the existing values will be updated. Nodes that can be deleted and are specified as not existing in the array, i.e. with value C_NOEXISTS, will be deleted if they existed before the call.

For a list entry, since the key values must be present in the array, it is not required that the key values are included in the path given by *`fmt`*. If the key values *are* included in the path, the key values in the array are ignored.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_INUSE

```
int maapi_set_values(int sock, int thandle, const confd_tag_value_t
*values, int n, const char *fmt, ...);
```

Set arbitrary sub-elements of a container or list entry. The *values* array must be populated with *n* values according to the specification of the *Tagged Value Array* format in the *XML STRUCTURES* section of the confd_types(3) manual page.

If the container or list entry itself, or any sub-elements that are specified as existing, do not exist before this call, they will be created, otherwise the existing values will be updated. Both mandatory and optional elements may be omitted from the array, and all omitted elements are left unchanged. To actually delete a non-mandatory leaf or presence container as described for `maapi_set_object()`, it may (as an extension of the format) be specified as C_NOEXISTS instead of being omitted.

For a list entry, the key values can be specified either in the path or via key elements in the array - if the values are in the path, the key elements can be omitted from the array. For sub-lists present in the array, the key elements must of course always also be present though, immediately following the C_XMLBEGIN element and in the order defined by the data model. It is also possible to delete a list entry by using a C_XMLBEGINDEL element, followed by the keys in data model order, followed by a C_XMLEND element.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_INUSE

`int` **`maapi_get_case`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`choice`*`, confd_value_t *`*`rcase`*`, const char *`*`fmt`*`, ...);`

When we use the YANG `choice` statement in the data model, this function can be used to find the currently selected `case`, avoiding useless `maapi_get_elem()` etc requests for nodes that belong to other cases. The *`fmt, ...`* arguments give the path to the list entry or container where the choice is defined, and *`choice`* is the name of the choice. The case value is returned to the confd_value_t that *`rcase`* points to, as type C_XMLTAG - i.e. we can use the `CONFD_GET_XMLTAG()` macro to retrieve the hashed tag value.

If we have "nested" choices, i.e. multiple levels of `choice` statements without intervening `container` or `list` statements in the data model, the *`choice`* argument must give a '/'-separated path with alternating choice and case names, from the data node given by the *`fmt, ...`* arguments to the specific choice that the request pertains to.

For a choice without a `mandatory true` statement where no case is currently selected, the function will fail with CONFD_ERR_NOEXISTS if the choice doesn't have a default case. If it has a default case, it will be returned unless the MAAPI_FLAG_NO_DEFAULTS flag is in effect (see `maapi_set_flags()` below) - if the flag is set, the value returned via *`rcase`* will have type C_DEFAULT.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED

`int` **`maapi_get_attrs`**`(int` *`sock`*`, int` *`thandle`*`, u_int32_t *`*`attrs`*`, int` *`num_attrs`*`, confd_attr_value_t **`*`attr_vals`*`, int *`*`num_vals`*`, const char *`*`fmt`*`, ...);`

Retrieve attributes for a configuration node. These attributes are currently supported:

```
/* CONFD_ATTR_TAGS: value is C_LIST of C_BUF/C_STR */
#define CONFD_ATTR_TAGS       0x80000000
/* CONFD_ATTR_ANNOTATION: value is C_BUF/C_STR */
#define CONFD_ATTR_ANNOTATION 0x80000001
/* CONFD_ATTR_INACTIVE: value is C_BOOL 1 (i.e. "true") */
#define CONFD_ATTR_INACTIVE   0x00000000
```

The *attrs* parameter is an array of attributes of length *num_attrs*, specifying the wanted attributes - if *num_attrs* is 0, all attributes are retrieved. If no attributes are found, **num_vals* is set to 0, otherwise an array of confd_attr_value_t elements is allocated and populated, its address stored in **attr_vals*, and **num_vals* is set to the number of elements in the array. The confd_attr_value_t struct is defined as:

```
typedef struct confd_attr_value {
    u_int32_t attr;
    confd_value_t v;
} confd_attr_value_t;
```

If any attribute values are returned (**num_vals* > 0), the caller must free the allocated memory by calling `confd_free_value()` for each of the confd_value_t elements, and `free(3)` for the **attr_vals* array itself.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_UNAVAILABLE

`int` **`maapi_set_attr`**`(int` *`sock`*`, int` *`thandle`*`, u_int32_t` *`attr`*`, confd_value_t *`*`v`*`, const char *`*`fmt`*`, ...);`

Set an attribute for a configuration node. See `maapi_get_attrs()` above for the supported attributes. To delete an attribute, call the function with a value of type C_NOEXISTS.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_BADTYPE, CONFD_ERR_NOEXISTS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_UNAVAILABLE

`int` **`maapi_delete_all`**`(int` *`sock`*`, int` *`thandle`*`, enum maapi_delete_how` *`how`*`);`

This function can be used to delete "all" the configuration data within a transaction. The *how* argument specifies the extent of "all":

`MAAPI_DEL_SAFE` Delete everything except namespaces that were exported to none (with `tailf:export none`). Toplevel nodes that cannot be deleted due to AAA rules are silently left in place, but descendant nodes will still be deleted if the AAA rules allow it.

`MAAPI_DEL_EXPORTED` Delete everything except namespaces that were exported to none (with `tailf:export none`). AAA rules are ignored, i.e. nodes are deleted even if the AAA rules don't allow it.

`MAAPI_DEL_ALL` Delete everything. AAA rules are ignored.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

`int` **`maapi_revert`**`(int` *`sock`*`, int` *`thandle`*`);`

This function removes all changes done to the transaction.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

`int` **`maapi_set_flags`**`(int` *`sock`*`, int` *`thandle`*`, int` *`flags`*`);`

We can modify some aspects of the read/write session by calling this function - these values can be used for the *flags* argument (ORed together if more than one) with this function and/or with maapi_start_trans_flags():

```
#define MAAPI_FLAG_HINT_BULK     (1 << 0)
#define MAAPI_FLAG_NO_DEFAULTS   (1 << 1)
#define MAAPI_FLAG_CONFIG_ONLY   (1 << 2)
#define MAAPI_FLAG_HIDE_INACTIVE (1 << 3) /* maapi_start_trans_flags() only */
#define MAAPI_FLAG_DELAYED_WHEN  (1 << 6) /* maapi_start_trans_flags() only */
```

MAAPI_FLAG_HINT_BULK tells the ConfD backplane that we will be reading substantial amounts of data. This has the effect that the get_object() and get_next_object() callbacks (if available) are used towards external data providers when we call maapi_get_elem() etc and maapi_get_next(). The maapi_get_object() function always operates as if this flag was set.

MAAPI_FLAG_NO_DEFAULTS says that we want to be informed when we read leafs with default values that have not had a value set. This is indicated by the returned value being of type C_DEFAULT instead of the actual value. The default value for such leafs can be obtained from the confd_cs_node tree provided by the library (see confd_types(3)).

MAAPI_FLAG_CONFIG_ONLY will make the maapi_get_xxx() functions return config nodes only - if we attempt to read operational data, it will be treated as if the nodes did not exist. This is mainly useful in conjunction with maapi_get_object() and list entries or containers that have both config and operational data (the operational data nodes in the returned array will have the "value" C_NOEXISTS), but the other functions also obey the flag.

MAAPI_FLAG_HIDE_INACTIVE can only be used with maapi_start_trans_flags(), and only when starting a readonly transaction (parameter *readwrite* == CONFD_READ). It will hide configuration data that has the CONFD_ATTR_INACTIVE attribute set, i.e. it will appear as if that data does not exist.

MAAPI_FLAG_DELAYED_WHEN can also only be used with maapi_start_trans_flags(), but regardless of whether the flag is used or not, the "delayed when" mode can subsequently be changed with maapi_set_delayed_when(). The flag is only meaningful when starting a read-write transaction (parameter *readwrite* == CONFD_READ_WRITE), and will cause "delayed when" mode to be enabled from the beginning of the transaction. See the description of maapi_set_delayed_when() for information about the "delayed when" mode.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

int **maapi_set_delayed_when**(int *sock*, int *thandle*, int *on*);

This function enables (*on* non-zero) or disables (*on* == 0) the "delayed when" mode of a transaction. When successful, it returns 1 or 0 as indication of whether "delayed when" was enabled or disabled before the call. See also the MAAPI_FLAG_DELAYED_WHEN flag for maapi_start_trans_flags().

The YANG when statement makes its parent data definition statement conditional. This can be problematic in cases where we don't have control over the order of writing different data nodes. E.g. when loading configuration from a file, the data that will satisfy the when condition may occur after the data that the when applies to, making it impossible to actually write the latter data into the transaction - since the when isn't satisfied, the data nodes effectively do not exist in the schema.

This is addressed by the "delayed when" mode for a transaction. When "delayed when" is enabled, it is possible to write to data nodes even though they are conditional on a when that isn't satisfied. It has no effect on reading though - trying to read data that is conditional on an unsatisfied when will always result in CONFD_ERR_NOEXISTS or equivalent. When disabling "delayed when", any "delayed" when

statements will take effect immediately - i.e. if the `when` isn't satisfied at that point, the conditional nodes and any data values for them will be deleted. If we don't explicitly disable "delayed when" by calling this function, it will be automatically disabled when the transaction enters the VALIDATE state (e.g. due to call of `maapi_apply_trans()`).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_NOEXISTS

NCS SPECIFIC FUNCTIONS

The functions in this sections can only be used with NCS, and specifically the maapi_shared_xxx() functions must be used for NCS FASTMAP, i.e. in the service `create()` callback. Those functions maintain attributes that are necessary when multiple service instances modify the same data.

`int` **`maapi_shared_create`**`(int` *`sock`*`, int` *`thandle`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

FASTMAP version of `maapi_create()`. Normally the *`flags`* parameter should be given as 0, but it is possible to suppress the creation of backpointer attributes by passing `MAAPI_SHARED_NO_BACKPOINTER` for *`flags`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOTCREATABLE, CONFD_ERR_INUSE

`int` **`maapi_shared_set_elem`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t *`*`v`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

`int` **`maapi_shared_set_elem2`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`strval`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

FASTMAP versions of `maapi_set_elem()` and `maapi_set_elem2()`. The *`flags`* parameter is currently unused and should be given as 0.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_INUSE

`int` **`maapi_shared_insert`**`(int` *`sock`*`, int` *`thandle`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

FASTMAP version of `maapi_insert()`. Normally the *`flags`* parameter should be given as 0, but it is possible to suppress the creation of backpointer attributes by passing `MAAPI_SHARED_NO_BACKPOINTER` for *`flags`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTDELETABLE

`int` **`maapi_shared_copy_tree`**`(int` *`sock`*`, int` *`thandle`*`, int` *`flags`*`, const char *`*`from`*`, const char *`*`tofmt`*`, ...);`

FASTMAP version of `maapi_copy_tree()`. Normally the *`flags`* parameter should be given as 0, but it is possible to suppress the creation of backpointer attributes by passing `MAAPI_SHARED_NO_BACKPOINTER` for *`flags`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_BADPATH

`int` **`maapi_ncs_apply_template`**`(int` *`sock`*`, int` *`thandle`*`, char *`*`template_name`*`, const struct ncs_name_value *`*`variables`*`, int` *`num_variables`*`, int` *`flags`*`, const char *`*`rootfmt`*`, ...);`

Apply a template that has been loaded into NCS. The *`template_name`* parameter gives the name of the template. The *`variables`* parameter is an *`num_variables`* long array of variables and names for substitution into the template. The struct ncs_name_value is defined as:

```
struct ncs_name_value {
    char *name;
    char *value;
};
```

The *`flags`* parameter is currently unused and should be given as 0.

Note

This is *not* a FASTMAP function - use `maapi_shared_ncs_apply_template()` for FASTMAP.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_XPATH

`int` **`maapi_shared_ncs_apply_template`**`(int` *`sock`*`, int` *`thandle`*`, char *`*`template_name`*`, const struct ncs_name_value *`*`variables`*`, int` *`num_variables`*`, int` *`flags`*`, const char *`*`rootfmt`*`, ...);`

FASTMAP version of `maapi_ncs_apply_template()`. Normally the *`flags`* parameter should be given as 0, but it is possible to suppress the creation of backpointer attributes by passing `MAAPI_SHARED_NO_BACKPOINTER` for *`flags`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_XPATH

`int` **`maapi_ncs_get_templates`**`(int` *`sock`*`, char ***`*`templates`*`, int *`*`num_templates`*`);`

Retrieve a list of the templates currently loaded into NCS. On success, a pointer to an array of template names is stored in *`templates`* and the length of the array is stored in *`num_templates`*. The library allocates memory for the result, and the caller is responsible for freeing it. This can in all cases be done with code like this:

```
char **templates;
int num_templates, i;

if (maapi_ncs_get_templates(sock, &templates, &num_templates) == CONFD_OK) {
    ...
    for (i = 0; i < num_templates; i++) {
        free(templates[i]);
    }
    if (num_templates > 0) {
        free(templates);
    }
```

```
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

MISCELLANEOUS FUNCTIONS

`int` **`maapi_delete_config`**`(int` *`sock`*`, enum confd_dbname` *`name`*`);`

This function empties a data store.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_EXTERNAL

`int` **`maapi_copy`**`(int` *`sock`*`, int` *`from_thandle`*`, int` *`to_thandle`*`);`

If we open two transactions from the same user session but towards different data stores, such as one transaction towards startup and one towards running, we can copy all data from one data store to the other with this function. This is a replace operation - any configuration that exists in the transaction given by *`to_handle`* but not in the one given by *`from_handle`* will be deleted from the *`to_handle`* transaction.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE

`int` **`maapi_copy_path`**`(int` *`sock`*`, int` *`from_thandle`*`, int` *`to_thandle`*`, const char *`*`fmt`*`, ...);`

Similar to `maapi_copy()`, but does a replacing copy only of the subtree rooted at the path given by *`fmt`* and remaining arguments.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE

`int` **`maapi_copy_tree`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`from`*`, const char *`*`tofmt`*`, ...);`

This function copies the entire configuration tree rooted at *`from`* to *`tofmt`*. List entries are created accordingly. If the destination already exists, *`from`* is copied on top of the destination. This function is typically used inside actions where we for example could use `maapi_copy_tree()` to copy a template configuration into a new list entry. The *`from`* path must be pre-formatted, e.g. using `confd_format_keypath()`, whereas the destination path is formatted by this function.

Note

The data models for the source and destination trees must match - i.e. they must either be identical, or the data model for the source tree must be a proper subset of the data model for the destination tree. This is always fulfilled when copying from one entry to another in a list, or if both source and destination tree have been defined via YANG `uses` statements referencing the same `grouping` definition. If a data model mismatch is detected, e.g. an existing data node in the source tree does not exist in the destination data model, or an existing leaf in the source tree has a value that is incompatible with the type of the leaf in the destination data model, `maapi_copy_tree()` will return CONFD_ERR with `confd_errno` set to CONFD_ERR_BADPATH.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_BADPATH

`int` **`maapi_insert`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`fmt`*`, ...);`

This function inserts a new entry in a list that uses the `tailf:indexed-view` statement. The key must be of type integer. If the inserted entry already exists, the existing and subsequent entries will be renumbered as needed, unless renumbering would require an entry to have a key value that is outside the range of the type for the key. In that case, the function returns CONFD_ERR with `confd_errno` set to CONFD_ERR_BADTYPE.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_BADTYPE, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTDELETABLE

`int` **`maapi_move`**`(int` *`sock`*`, int` *`thandle`*`, confd_value_t*` *`tokey`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

This function moves an existing list entry, i.e. renames the entry using the *`tokey`* parameter, which is an array containing *`n`* keys.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTMOVABLE, CONFD_ERR_ALREADY_EXISTS

`int` **`maapi_move_ordered`**`(int` *`sock`*`, int` *`thandle`*`, enum maapi_move_where` *`where`*`, confd_value_t*` *`tokey`*`, int` *`n`*`, const char *`*`fmt`*`, ...);`

For a list with the YANG `ordered-by user` statement, this function can be used to change the order of entries, by moving one entry to a new position. When new entries in such a list are created with `maapi_create()`, they are always placed last in the list. The path given by *`fmt`* and the remaining arguments identifies the entry to move, and the new position is given by the *`where`* argument:

MAAPI_MOVE_FIRST Move the entry first in the list. The *`tokey`* and *`n`* arguments are ignored, and can be given as NULL and 0.

MAAPI_MOVE_LAST Move the entry last in the list. The *`tokey`* and *`n`* arguments are ignored, and can be given as NULL and 0.

MAAPI_MOVE_BEFORE Move the entry to the position before the entry given by the *`tokey`* argument, which is an array of key values with length *`n`*.

MAAPI_MOVE_AFTER Move the entry to the position after the entry given by the *`tokey`* argument, which is an array of key values with length *`n`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_NOT_WRITABLE, CONFD_ERR_NOEXISTS, CONFD_ERR_NOTMOVABLE

`int` **`maapi_authenticate`**`(int` *`sock`*`, const char *`*`user`*`, const char *`*`pass`*`, char *`*`groups[]`*`, int` *`n`*`);`

If we are implementing a proprietary management agent with MAAPI API, the function `maapi_start_user_session()` requires the application to tell ConfD which groups the user are member of. ConfD itself has the capability to authenticate users. A MAAPI application can use `maapi_authenticate()` to let ConfD authenticate the user, as per the AAA configuration in confd.conf

If the authentication is successful, the function returns `1`, and the *`groups[]`* array is populated with at most *`n-1`* NUL-terminated strings containing the group names, followed by a NULL pointer that indicates the end of the group list. The strings are dynamically allocated, and it is up to the caller to free the memory

by calling `free(3)` for each string. If the function is used in a context where the group names are not needed, pass `1` for the *`n`* parameter.

If the authentication fails, the function returns `0`, and `confd_lasterr()` (see confd_lib_lib(3)) will return a message describing the reason for the failure.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION

`int` **`maapi_authenticate2`**`(int` *`sock`*`, const char *`*`user`*`, const char *`*`pass`*`, const struct confd_ip *`*`src_addr`*`, int` *`src_port`*`, const char *`*`context`*`, enum confd_proto` *`prot`*`, char *`*`groups[]`*`, int` *`n`*`);`

This function does the same thing as `maapi_authenticate()`, but allows for passing of the additional parameters *`src_addr`*, *`src_port`*, *`context`*, and *`prot`*, which otherwise are passed only to `maapi_start_user_session()`/`maapi_start_user_session2()`. These parameters are not used when ConfD performs the authentication, but they will be passed to an external authentication executable (see the External authentication section of the AAA chapter in the User Guide) if `/confd-Config/aaa/externalAuthentication/includeExtra` is set to "true" in `confd.conf`, see confd.conf(5). They will also be made available to the authentication callback that can be registered by an application (see confd_lib_dp(3)).

Errors: CONFD_ERR_PROTOUSAGE, CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION

`int` **`maapi_attach`**`(int` *`sock`*`, int` *`hashed_ns`*`, struct confd_trans_ctx *`*`ctx`*`);`

While ConfD is executing a transaction, we have a number of situations where we wish to invoke user C code which can interact in the transaction. One such situation is when we wish to write semantic validation code which is invoked in the validation phase of a ConfD transaction. This code needs to execute within the context of the executing transaction, it must thus have access to the "shadow" storage where all not-yet-committed data is kept.

This function attaches to a existing transaction. See the Semantic Validation chapter in the User Guide for example code.

Another situation where we wish to attach to the executing transaction is when we are using the notifications API and subscribe to notification of type CONFD_NOTIF_COMMIT_DIFF and wish to read the committed diffs from the transaction.

The *`hashed_ns`* parameter is basically just there to save a call to `maapi_set_namespace()`. We can call `maapi_set_namespace()` any number of times to change from the one we passed to `maapi_attach()`, and we can also give the namespace in prefix form in the path parameter to the read/write functions - see the `maapi_set_namespace()` description.

If we do not want to give a specific namespace when invoking `maapi_attach()`, we can give 0 for the *`hashed_ns`* parameter (-1 works too but is deprecated). We can still call the read/write functions as long as the toplevel element in the path is unique, but otherwise we must call `maapi_set_namespace()`, or use a prefix in the path.

`int` **`maapi_attach2`**`(int` *`sock`*`, int` *`hashed_ns`*`, int` *`usid`*`, int` *`thandle`*`);`

When we write proprietary CLI commands in C and we wish those CLI commands to be able to use MAAPI to read and write data inside the same transaction the CLI command was invoked in, we do not have an initialized transaction structure available. Then we must use this function. CLI commands get the *`usid`* passed in UNIX environment variable `CONFD_MAAPI_USID` and the *`thandle`* passed in environment variable `CONFD_MAAPI_THANDLE`. We also need to use this function when implementing such CLI commands via action `command()` callbacks, see the confd_lib_dp(3) man page. In this case the *`usid`* is

provided via `uinfo->usid` and the *`thandle`* via `uinfo->actx.thandle`. To use the user session id that is the owner of the transaction, set *`usid`* to 0. If the namespace does not matter set *`hashed_ns`* to 0, see `maapi_attach()`.

`int` **`maapi_attach_init`**`(int` *`sock`*`, int *`*`thandle`*`);`

This function is used to attach the MAAPI socket to the special transaction available in phase0 used for CDB initialization and upgrade. The function is also used if we need to modify CDB data during in-service data model upgrade (see the "In-service Data Model Upgrade" chapter in the User Guide). The transaction handle, which is used in subsequent calls to MAAPI, is filled in by the function upon successful return. See the CDB chapter in the User Guide.

`int` **`maapi_detach`**`(int` *`sock`*`, struct confd_trans_ctx *`*`ctx`*`);`

Detaches an attached MAAPI socket. This function is typically called in the `stop()` callback in validation code. An attached MAAPI socket will be automatically detached when the ConfD transaction terminates. This function performs an explicit detach.

`int` **`maapi_detach2`**`(int` *`sock`*`, int` *`thandle`*`);`

Detaches an attached MAAPI socket when we do not have an initialized transaction structure available, see `maapi_attach2()` above. This is mainly useful in an action `command()` callback.

`int` **`maapi_diff_iterate`**`(int` *`sock`*`, int` *`thandle`*`, enum maapi_iter_ret (*`*`iter`*`)(confd_hkeypath_t *`*`kp`*`, enum maapi_iter_op` *`op`*`, confd_value_t *`*`oldv`*`, confd_value_t *`*`newv`*`, void *`*`state`*`), int` *`flags`*`, void *`*`init_state`*`);`

This function can be called from an attached MAAPI session. The purpose of the function is to iterate through the transaction diff. It can typically be used in conjunction with the notification API when we subscribe to CONFD_NOTIF_COMMIT_DIFF events. It can also be used inside validation callbacks.

For all diffs in the transaction the supplied callback function `iter()` will be called. The `iter()` callback receives the *`confd_hkeypath_t kp`* which uniquely identifies which node in the data tree that is affected, the operation, and an optional value. The *`op`* parameter gives the modification as:

MOP_CREATED — The list entry, `presence` container, or leaf of type `empty` given by *`kp`* has been created.

MOP_DELETED — The list entry, `presence` container, or optional leaf given by *`kp`* has been deleted.

MOP_MODIFIED — A descendant of the list entry given by *`kp`* has been modified.

MOP_VALUE_SET — The value of the leaf given by *`kp`* has been set to *`newv`*. If the MAAPI_FLAG_NO_DEFAULTS flag has been set and the default value for the leaf has come into effect, *`newv`* will be of type C_DEFAULT instead of giving the default value.

MOP_MOVED_AFTER — The list entry given by *`kp`*, in an `ordered-by user` list, has been moved. If *`newv`* is NULL, the entry has been moved first in the list, otherwise it has been moved after the entry given by *`newv`*. In this case *`newv`* is a pointer to an array of key values identifying an entry in the list. The array is terminated with an element that has type C_NOEXISTS.

MOP_ATTR_SET — An attribute for the node given by *`kp`* has been modified (see the description of `maapi_get_attrs()` for the supported attributes). The `iter()` callback will only get this invocation when attributes are enabled in `confd.conf`

(/confdConfig/enableAttributes, see confd.conf(5)) *and* the flag ITER_WANT_ATTR has been passed to maapi_diff_iterate(). The *newv* parameter is a pointer to a 2-element array, where the first element is the attribute represented as a confd_value_t of type C_UINT32 and the second element is the value the attribute was set to. If the attribute has been deleted, the second element is of type C_NOEXISTS.

The *oldv* parameter passed to iter() is always NULL.

If iter() returns ITER_STOP, no more iteration is done, and CONFD_OK is returned. If iter() returns ITER_RECURSE iteration continues with all children to the node. If iter() returns ITER_CONTINUE iteration ignores the children to the node (if any), and continues with the node's sibling.

The *flags* parameter is a bitmask with the following bits:

ITER_WANT_ATTR Enable MOP_ATTR_SET invocations of the iter() function.

ITER_WANT_P_CONTAINER Invoke iter() for modified presence-containers.

The *state* parameter can be used for any user supplied state (i.e. whatever is supplied as *init_state* is passed as *state* to iter() in each invocation).

The iter() invocations are not subjected to AAA checks, i.e. regardless of which path we have and which context was used to create the MAAPI socket, all changes are provided.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS, CONFD_ERR_BADSTATE.

CONFD_ERR_BADSTATE is returned when we try to iterate on a transaction which is in the wrong state and not attached.

int **maapi_keypath_diff_iterate**(int *sock*, int *thandle*, enum maapi_iter_ret (**iter*)(confd_hkeypath_t **kp*, enum maapi_iter_op *op*, confd_value_t **oldv*, confd_value_t **newv*, void **state*), int *flags*, void **initstate*, const char **fmtpath*, ...);

This function behaves precisely like the maapi_diff_iterate() function except that it takes an additional format path argument. This path prunes the diff and only changes below the provided path are considered.

int **maapi_iterate**(int *sock*, int *thandle*, enum maapi_iter_ret (**iter*)(confd_hkeypath_t **kp*, confd_value_t **v*, confd_attr_value_t **attr_vals*, int *num_attr_vals*, void **state*), int *flags*, void **initstate*, const char **fmtpath*, ...);

This function can be used to iterate over all the data in a transaction and the underlying data store, as opposed to iterating over only the changes like maapi_diff_iterate() and maapi_keypath_diff_iterate() do. The *fmtpath* parameter can be used to prune the iteration to cover only the subtree below the given path, similar to maapi_keypath_diff_iterate() - if *fmtpath* is given as "/", there will not be any such pruning. Additionally, if the flag MAAPI_FLAG_CONFIG_ONLY is in effect (see maapi_set_flags()), all operational data subtrees will be excluded from the iteration. The *flags* parameter can be given as ITER_WANT_ATTR to request attribute values, otherwise it should be 0.

The supplied callback function iter() will be called for each node in the data tree included in the iteration. It receives the *kp* parameter which uniquely identifies the node, and if the node is a leaf with a type, also the value of the leaf as the *v* parameter - otherwise *v* is NULL. If the flag

ITER_WANT_ATTR was given in the call of maapi_iterate(), and the node has any attributes set, the *attr_vals* will point to a *num_attr_vals* long array of attributes and values (see maapi_get_attrs()), otherwise *attr_vals* is NULL. The return value from iter() has the same effect as for maapi_diff_iterate().

int **maapi_get_running_db_status**(int *sock*);

If a transaction fails in the commit() phase, the configuration database is in in a possibly inconsistent state. This function queries ConfD on the consistency state. Returns 1 if the configuration is consistent and 0 otherwise.

int **maapi_set_running_db_status**(int *sock*, int *status*);

This function explicitly sets ConfDs notion of the consistency state.

int **maapi_list_rollbacks**(int *sock*, struct maapi_rollback *rp, int *rp_size);

List at most *rp_size number of rollback files. The number of existing rollback files is reported in *rp_size as well. The function will populate an array of maapi_rollback structs.

int **maapi_load_rollback**(int *sock*, int *thandle*, int *rollback_num*);

Install a rollback file.

int **maapi_request_action**(int *sock*, confd_tag_value_t *params, int *nparams*, confd_tag_value_t **values, int *nvalues, int *hashed_ns*, const char *fmt, ...);

Invoke an action defined in the data model. The *params* and *values* arrays are the parameters for and results from the action, respectively, and use the Tagged Value Array format described in the XML STRUCTURES section of the confd_types(3) manual page. The library allocates memory for the result values, and the caller is responsible for freeing it. This can in all cases be done with code like this:

```
confd_tag_value_t *values;
int nvalues = 0, i;

if (maapi_request_action(sock, params, nparams,
                         &values, &nvalues, myprefix__ns,
                         "/path/to/action") == CONFD_OK) {
    ...
    for (i = 0; i < nvalues; i++)
        confd_free_value(CONFD_GET_TAG_VALUE(&values[i]));
    if (nvalues > 0)
        free(values);
}
```

However if the value array is known not to include types that require memory allocation (see maapi_get_elem() above), only the array itself needs to be freed.

The socket must have an established user session. The path given by *fmt* and the varargs list is the full path to the action, i.e. the final element must be the name of the action in the data model. Since actions are not associated with ConfD transactions, the namespace must be provided and the path must be absolute - but see maapi_request_action_th() below.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

`int` **`maapi_request_action_th`**`(int sock, int thandle, confd_tag_value_t *params, int nparams, confd_tag_value_t **values, int *nvalues, const char *fmt, ...);`

Does the same thing as `maapi_request_action()`, but uses the current namespace, the path position, and the user session from the transaction indicated by *`thandle`*, and makes the transaction handle available to the action() callback, see confd_lib_dp(3) (this is the only relation to the transaction, and the transaction is not affected in any way by the call itself). This function may be convenient in some cases where actions are invoked in conjunction with a transaction, and it must be used if the action needs to access the transaction store.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

`int` **`maapi_request_action_str_th`**`(int sock, int thandle, char **output, const char *cmd_fmt, const char *path_fmt, ...);`

Does the same thing as `maapi_request_action_th()`, but takes the parameters as a string and returns the result as a string. The library allocates memory for the result string, and the caller is responsible for freeing it. This can in all cases be done with code like this:

```
char *output = NULL;

if (maapi_request_action_str_th(sock, th, &output,
    "test reverse listint [ 1 2 3 4 ]", "/path/to/action") == CONFD_OK) {
    ...
    free(output);
}
```

The varargs in the end of the function must contain all values listed in both format strings (that is *`cmd_fmt`* and *`path_fmt`*) in the same order as they occur in the strings. Here follows an equivalent example which uses the format strings:

```
char *output = NULL;

if (maapi_request_action_str_th(sock, th, &output,
    "test %s [ 1 2 3 %d ]", "%s/action",
    "reverse listint", 4, "/path/to") == CONFD_OK) {
    ...
    free(output);
}
```

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOSESSION, CONFD_ERR_BADPATH, CONFD_ERR_NOEXISTS, CONFD_ERR_BADTYPE, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

`int` **`maapi_xpath2kpath`**`(int sock, const char *xpath, confd_hkeypath_t **hkp);`

Convert a XPath path to a hashed keypath. The XPath expression must be an "instance identifier", i.e. all elements and keys must be fully specified. Namespace prefixes are optional, unless required to resolve ambiguities (e.g. when multiple namespaces have the same root element).

The returned keypath is dynamically allocated, and may further contain dynamically allocated elements. The caller must free the allocated memory, easiest done by calling `confd_free_hkeypath()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

```
int maapi_user_message(int sock, const char *to, const char *message,
const char *sender);
```

Send a message to a specific user, a specific user session or all users depending on the *to* parameter. If set to a user name, then *message* will be delivered to all CLI and Web UI sessions by that user. If set to an integer string, eg "10", then *message* will be delivered to that specific user session, CLI or Web UI. If set to "all" then all users will get the *message*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_sys_message(int sock, const char *to, const char *message);
```

Send a message to a specific user, a specific user session or all users depending on the *to* parameter. If set to a user name, then *message* will be delivered to all CLI and Web UI sessions by that user. If set to an integer string, eg "10", then *message* will be delivered to that specific user session, CLI or Web UI. If set to "all" then all users will get the *message*. No formatting of the message is performed as opposed to the user message where a timestamp and sender information is added to the message.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_prio_message(int sock, const char *to, const char *message);
```

Send a high priority message to a specific user, a specific user session or all users depending on the *to* parameter. If set to a user name, then *message* will be delivered to all CLI and Web UI sessions by that user. If set to an integer string, eg "10", then *message* will be delivered to that specific user session, CLI or Web UI. If set to "all" then all users will get the *message*. No formatting of the message is performed as opposed to the user message where a timestamp and sender information is added to the message.

The message will not be delayed until the user terminates any ongoing command but will be output directly to the terminal without delay. Messages sent using the maapi_sys_message and maapi_user_message, on the other hand, are not displayed in the middle of some other output but delayed until the any ongoing commands have terminated.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_prompt(int sock, int usess, const char *prompt, int echo,
char *res, int size);
```

Prompt user for a string. The *echo* parameter is used to control if the input should be echoed or not. If set to CONFD_ECHO all input will be visible and if set to CONFD_NOECHO only stars will be shown instead of the actual characters entered by the user. The resulting string will be stored in *res* and it will be NUL terminated.

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_prompt2(int sock, int usess, const char *prompt, int echo,
int timeout, char *res, int size);
```

This function does the same as `maapi_cli_prompt()`, but also takes a *timeout* parameter, which controls how long (in seconds) to wait for input before aborting.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_prompt_oneof(int sock, int usess, const char *prompt, char
**choice, int count, char *res, int size);
```

Prompt user for one of the strings given in the *choice* parameter. For example:

```
int res;
char buf[BUFSIZ];
char *choice[] = {"yes","no"};

...

res = maapi_cli_prompt_oneof(sock, uinfo->usid,
                             "Do you want to proceed (yes/no): ",
                             choice, 2, buf, BUFSIZ);
```

The user can enter a unique prefix of the choice but the value returned in buf will always be one of the strings provided in the *choice* parameter. The result string stored in buf is NUL terminated. If the user enters a value not in *choice* he will automatically be re-prompted. For example:

```
Do you want to proceed (yes/no): maybe
The value must be one of: yes,no.
Do you want to proceed (yes/no):
```

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_prompt_oneof2`**`(int` *`sock`*`, int` *`usess`*`, const char *`*`prompt`*`, char **`*`choice`*`, int` *`count`*`, int` *`timeout`*`, char *`*`res`*`, int` *`size`*`);`

This function does the same as `maapi_cli_promt_oneof()`, but also takes a *`timeout`* parameter. If no activity is seen for *`timeout`* seconds an error is returned.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_read_eof`**`(int` *`sock`*`, int` *`usess`*`, int` *`echo`*`, char *`*`res`*`, int` *`size`*`);`

Read a multi line string from the CLI. The user has to end the input using ctrl-D. The entered characters will be stored NUL terminated in res. The *`echo`* parameters controls if the entered characters should be echoed or not. If set to CONFD_ECHO they will be visible and if set to CONFD_NOECHO stars will be echoed instead.

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_read_eof2`**`(int` *`sock`*`, int` *`usess`*`, int` *`echo`*`, int` *`timeout`*`, char *`*`res`*`, int` *`size`*`);`

This function does the same as `maapi_cli_read_eof()`, but also takes a *`timeout`* parameter, which indicates how long the user may be idle (in seconds) before the reading is aborted.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_write`**`(int` *`sock`*`, int` *`usess`*`, const char *`*`buf`*`, int` *`size`*`);`

Write to the CLI.

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_printf`**`(int` *`sock`*`, int` *`usess`*`, const char *`*`fmt`*`, ...);`

Write to the CLI using printf formatting. This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_vprintf`**`(int` *`sock`*`, int` *`usess`*`, const char *`*`fmt`*`, va_list` *`args`*`);`

Does the same as `maapi_cli_printf()`, but takes a single va_list argument instead of a variable number of arguments, like `vprintf()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_accounting`**`(int` *`sock`*`, const char *`*`user`*`, const int` *`usid`*`, const char *`*`cmdstr`*`);`

Generate an audit log entry in the CLI audit log.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_diff_cmd`**`(int` *`sock`*`, int` *`thandle`*`, int` *`thandle_old`*`, char *`*`res`*`, int` *`size`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

Get the diff between two sessions as C-/I-style CLI commands.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_path_cmd`**`(int` *`sock`*`, int` *`thandle`*`, char *`*`res`*`, int` *`size`*`, int` *`flags`*`, const char *`*`fmt`*`, ...);`

This function tries to determine which C-/I-style CLI command can be associated with a given path in the data model in context of a given transaction. This is determined by running the formatting code used by the 'show running-config' command for the subtree given by the path, and the looking for text lines associated with the given path. Consequentcly, if the path does not exist in the transaction no output will be generated, or if tailf:cli- annotations have been used to suppress the 'show running-config' text for a path then no such command can be derived.

The *`flags`* can be given as `MAAPI_FLAG_EMIT_PARENTS` to enable the commands to reach the sub-mode for the path to be emitted.

The *`flags`* can be given as `MAAPI_FLAG_DELETE` to emit the command to delete the given path.

The *`flags`* can be given as `MAAPI_FLAG_NON_RECURSIVE` to prevent that all children to a container or list item are displayed.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_cli_cmd_to_path`**`(int` *`sock`*`, const char *`*`line`*`, char *`*`ns`*`, int` *`nsize`*`, char *`*`path`*`, int` *`psize`*`);`

Given a data model path formatted as a C- and I-style command, try to determine the corresponding name-space and path. If the string cannot be interpreted as a path an error message is given indicating that the

string is either an operational mode command, a configuration mode command, or just badly formatted. The string is interpreted in the context of the current running configuration, ie all xpath expressions in the data model are evaluated in the context of the running config. Note that the same input may result in a correct answer when invoked with one state of the running config, and an error if the running config has another state due to different list elements being present, or xpath (when and display-when) expressions are being evaluated differently.

This function requires that the socket has an established user session.

The *line* is the NUL terminated string of command tokens to be interpreted.

The *ns* and *path* parameters are used for storing the resulting namespace and path.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd_to_path2(int sock, int thandle, const char *line, char
*ns, int nsize, char *path, int psize);
```

Given a data model path formatted as a C- and I-style command, try to determine the corresponding namespace and path. If the string cannot be interpreted as a path an error message is given indicating that the string is either an operational mode command, a configuration mode command, or just badly formatted. The string is interpreted in the context of the provided transaction handler, ie all xpath expressions in the data model are evaluated in the context of the transaction. Note that the same input may result in a correct answer when invoked with one state of one config, and an error when given another config due to different list elements being present, or xpath (when and display-when) expressions are being evaluated differently.

This function requires that the socket has an established user session.

The *th* is a transaction handler.

The *line* is the NUL terminated string of command tokens to be interpreted.

The *ns* and *path* parameters are used for storing the resulting namespace and path.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd(int sock, int usess, const char *buf, int size);
```

Execute CLI command in ongoing CLI session.

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd2(int sock, int usess, const char *buf, int size, int
flags);
```

Execute CLI command in ongoing CLI session.

This function is intended to be called from inside an action callback when invoked from the CLI. The flags field is used to disable certain checks during the execution. The value is a bitmask.

MAAPI_CMD_NO_FULLPATHDo not perform the fullpath check on show commands.

MAAPI_CMD_NO_HIDDEN Allows execution of hidden CLI commands.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd3(int sock, int usess, const char *buf, int size, int
flags, const char *unhide, int usize);
```

Execute CLI command in ongoing CLI session.

This function is intended to be called from inside an action callback when invoked from the CLI. The flags field is used to disable certain checks during the execution. The value is a bitmask.

MAAPI_CMD_NO_FULLPATHDo not perform the fullpath check on show commands.

MAAPI_CMD_NO_HIDDEN Allows execution of hidden CLI commands.

The unhide parameter is used for passing a hide group which is unhidden during the execution of the command.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd4(int sock, int usess, const char *buf, int size, int
flags, char **unhide, int usize);
```

Execute CLI command in ongoing CLI session.

This function is intended to be called from inside an action callback when invoked from the CLI. The flags field is used to disable certain checks during the execution. The value is a bitmask.

MAAPI_CMD_NO_FULLPATHDo not perform the fullpath check on show commands.

MAAPI_CMD_NO_HIDDEN Allows execution of hidden CLI commands.

The unhide parameter is used for passing hide groups which are unhidden during the execution of the command.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd_io(int sock, int usess, const char *buf, int size,
int flags, const char *unhide, int usize);
```

Execute CLI command in ongoing CLI session and output result on socket.

This function is intended to be called from inside an action callback when invoked from the CLI. The flags field is used to disable certain checks during the execution. The value is a bitmask.

MAAPI_CMD_NO_FULLPATHDo not perform the fullpath check on show commands.

MAAPI_CMD_NO_HIDDEN Allows execution of hidden CLI commands.

The unhide parameter is used for passing a hide group which is unhidden during the execution of the command.

The function returns `CONFD_ERR` on error or a positive integer id that can subsequently be used together with `confd_stream_connect()`. ConfD will write all data in a stream on that socket and when done, ConfD will close its end of the socket.

Once the stream socket is connected we can read the output from the cli command data on the socket. We need to continue reading until we receive EOF on the socket. To check if the command was successful we use the function. `maapi_cli_cmd_io_result()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd_io2(int sock, int usess, const char *buf, int size,
int flags, char **unhide, int usize);
```

Execute CLI command in ongoing CLI session and output result on socket.

This function is intended to be called from inside an action callback when invoked from the CLI. The flags field is used to disable certain checks during the execution. The value is a bitmask.

MAAPI_CMD_NO_FULLPATHDo not perform the fullpath check on show commands.

MAAPI_CMD_NO_HIDDEN Allows execution of hidden CLI commands.

The unhide parameter is used for passing hide groups which are unhidden during the execution of the command.

The function returns `CONFD_ERR` on error or a positive integer id that can subsequently be used together with `confd_stream_connect()`. ConfD will write all data in a stream on that socket and when done, ConfD will close its end of the socket.

Once the stream socket is connected we can read the output from the cli command data on the socket. We need to continue reading until we receive EOF on the socket. To check if the command was successful we use the function. `maapi_cli_cmd_io_result()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_cmd_io_result(int sock, int id);
```

We use this function to read the status of executing a cli command and streaming the result over a socket. The *`sock`* parameter must be the same maapi socket we used for `maapi_cli_cmd_io()` and the *`id`* parameter is the *`id`* returned by `maapi_cli_cmd_io()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

```
int maapi_cli_get(int sock, int usess, const char *opt, char *res, int
size);
```

Read CLI session parameter or attribute.

This function is intended to be called from inside an action callback when invoked from the CLI.

Possible params are for C and I-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, output-file, screen-length, screen-width, history, terminal, autowizard, "service prompt config" , show-defaults, and if enabled, display-level. And for J-style: complete-on-space, idle-timeout, ignore-leading-space, paginate, "output file", "screen length", "screen width", terminal, history, autowizard, "show defaults", and if enabled, display-level. In addition to this the attributes called annotation, tags and inactive can be read.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_cli_set(int sock, int usess, const char *opt, const char
*value);
```

Set CLI session parameter.

This function is intended to be called from inside an action callback when invoked from the CLI.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_set_readonly_mode(int sock, int flag);
```

There are certain situations where we want to explicitly control if a ConfD instance should be able to handle write operations from the northbound agents. In certain high-availability scenarios we may want to ensure that a node is a true readonly node, i.e. it should not be possible to initiate new write transactions on that node.

It can also be interesting in upgrade scenarios where we are interested in making sure that no configuration changes can occur during some interval.

This function toggles the readonly mode of a ConfD instance. If the *flag* parameter is non-zero, ConfD will be set in readonly mode, if it is zero, ConfD will be taken out of readonly mode. It is also worth to note that when a ConfD HA node is in slave mode as instructed by the application, no write transactions can occur regardless of the value of the flag set by this function.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

```
int maapi_disconnect_remote(int sock, const char *address);
```

Disconnect all remote connections between `CONFD_IPC_PORT` (see the ConfD IPC section in the Advanced Topics chapter in the User Guide) and *address*.

Since ConfD clients, e.g. CDB readers/subscribers, are connected using TCP it is also possible to do this remotely over a network. However since TCP doesn't offer a fast and reliable way of detecting that the other end has disappeared ConfD can get stuck waiting for a reply from such a disconnected client.

In some environments there will be an alternative supervision method that can detect when a remote host is unavailable, and in that situation this function can be used to instruct ConfD to drop all remote connections to a particular host. The address parameter is an IP address as a string, and the socket is a maapi socket obtained using `maapi_connect()`. On success, the function returns the number of connections that were closed.

Note

ConfD will close all its sockets with remote address *address*, *except* HA connections. For HA use `confd_ha_slave_dead()` or an HA state transition.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE, CONFD_ERR_UNAVAILABLE

```
int maapi_disconnect_sockets(int sock, int *sockets, int nsocks);
```

This function is an alternative to `maapi_disconnect_remote()` that can be useful in particular when using the "External IPC" functionality (see "Using a different IPC mechanism" in the ConfD IPC section in the Advanced Topics chapter in the User Guide). In this case ConfD does not have any knowledge of the remote address of the IPC connections, and thus `maapi_disconnect_remote()` is not applicable. The `maapi_disconnect_sockets()` instead takes an array of *nsocks* socket file descriptor numbers for the *sockets* parameter.

ConfD will close all connected sockets whose local file descriptor number is included the *sockets* array. The file descriptor numbers can be obtained e.g. via the **lsof(8)** command, or some similar tool in case **lsof** does not support the IPC mechanism that is being used.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE

```
int maapi_save_config(int sock, int thandle, int flags, const char
*fmtpath, ...);
```

This function can be used to save the entire config (or a subset thereof) in different formats. The *flags* parameter controls the saving as follows. The value is a bitmask.

MAAPI_CONFIG_XML
The configuration format is XML.

MAAPI_CONFIG_XML_PRETTY
The configuration format is pretty printed XML.

MAAPI_CONFIG_JSON
The configuration is in JSON format.

MAAPI_CONFIG_J
The configuration is in curly bracket Juniper CLI format.

MAAPI_CONFIG_C
The configuration is in Cisco XR style format.

MAAPI_CONFIG_C_IOS
The configuration is in Cisco IOS style format.

MAAPI_CONFIG_XPATH
The *fmtpath* and remaining arguments give an XPath filter instead of a keypath. Can only be used with MAAPI_CONFIG_XML and MAAPI_CONFIG_XML_PRETTY.

MAAPI_CONFIG_WITH_DEFAULTS
Default values are part of the configuration dump.

MAAPI_CONFIG_SHOW_DEFAULTS
Default values are also shown next to the real configuration value. Applies only to the CLI formats.

MAAPI_CONFIG_WITH_OPER
Include operational data in the dump.

MAAPI_CONFIG_HIDE_ALL
Hide all hidden nodes (see below).

MAAPI_CONFIG_UNHIDE_ALL
Unhide all hidden nodes (see below).

MAAPI_CONFIG_WITH_SERVICE_META
Include NCS service-meta-data attributes (refcounter, backpointer, and original-value) in the dump.

MAAPI_CONFIG_NO_PARENTS
When a path is provided its parent nodes are by default included. With this option the output will begin immediately at path - skipping any parents.

The provided path indicates which part(s) of the configuration to save. By default it is interpreted as a keypath as for other MAAPI functions, and thus identifies the root of a subtree to save. However it is possible to indicate wildcarding of list keys by completely omitting key elements - i.e. this requests save of a subtree for each entry of the corresponding list. For MAAPI_CONFIG_XML and MAAPI_CONFIG_XML_PRETTY it is alternatively possible to give an XPath filter, by including the flag MAAPI_CONFIG_XPATH.

If for example *fmtpath* is "/aaa:aaa/authentication/users" we dump a subtree of the AAA data, while if it is "/aaa:aaa/authentication/users/user/homedir", we dump only the homedir leaf for each user in the AAA data. If *fmtpath* is NULL, the entire configuration is dumped, except that namespaces with restricted export (from tailf:export) are treated as follows:

- When the MAAPI_CONFIG_XML or MAAPI_CONFIG_XML_PRETTY formats are used, the context of the user session that started the transaction is used to select namespaces with restricted export. If the "system" context is used, all namespaces are selected, regardless of export restriction.
- When one of the CLI formats is used, the context used to select namespaces with restricted export is always "cli".

By default, the treatment of nodes with a tailf:hidden statement depends on the state of the transaction. For a transaction started via MAAPI, no nodes are hidden, while for a transaction started by another northbound agent (e.g. CLI) and attached to, the nodes that are hidden are the same as in that agent session. The default can be overridden by using one of the flags MAAPI_CONFIG_HIDE_ALL and MAAPI_CONFIG_UNHIDE_ALL.

The function returns CONFD_ERR on error or a positive integer id that can subsequently be used together with confd_stream_connect(). Thus this function doesn't save the configuration to a file, but rather it returns an integer than is used together with a ConfD stream socket. ConfD will write all data in a stream on that socket and when done, ConfD will close its end of the socket. Thus the following code snippet indicates the usage pattern of this function.

```
int id;
int streamsock;
struct sockaddr_in addr;

id = maapi_save_config(sock, th, flags, path);
if (id < 0) {
    ... handle error ...
}

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

streamsock = socket(PF_INET, SOCK_STREAM, 0);
confd_stream_connect(streamsock, (struct sockaddr*)&addr,
                     sizeof(struct sockaddr_in), id, 0);
```

Once the stream socket is connected we can read the configuration data on the socket. We need to continue reading until we receive EOF on the socket. To check if the configuration retrieval was successful we use the function maapi_save_config_result().

The stream socket must be connected within 10 seconds after the id is received.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BAD_TYPE

int **maapi_save_config_result**(int *sock*, int *id*);

We use this function to verify that we received the entire configuration over the stream socket. The *sock* parameter must be the same maapi socket we used for maapi_save_config() and the *id* parameter is the *id* returned by maapi_save_config().

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

int **maapi_load_config**(int *sock*, int *thandle*, int *flags*, const char **filename*);

This function loads a configuration from *filename* into ConfD. The *th* parameter is a transaction handle. This can be either for a transaction created by the application, in which case the application must also apply

the transaction, or for an attached transaction (which must not be applied by the application). The format of the file can be either XML, curly bracket Juniper CLI format, Cisco XR style format, or Cisco IOS style format. The caller of the function has to indicate which it is by using one of the MAAPI_CONFIG_XML, MAAPI_CONFIG_J, MAAPI_CONFIG_C, or MAAPI_CONFIG_C_IOS flags, with the same meanings as for maapi_save_config(). If the name of the file ends in .gz (or .Z) then the file is assumed to be gzipped, and will be uncompressed as it is loaded.

Note

If you use a relative pathname for *filename*, it is taken as relative to the working directory of the ConfD daemon, i.e. the directory where the daemon was started.

By default the complete configuration (as allowed by the user of the current transaction) is deleted before the file is loaded. To merge the contents of the file use the MAAPI_CONFIG_MERGE flag. To replace only the part of the configuration that is present in the file, use the MAAPI_CONFIG_REPLACE flag.

Additional flags for MAAPI_CONFIG_XML:

MAAPI_CONFIG_WITH_OPER
: Any operational data in the file should be ignored (instead of producing an error).

MAAPI_CONFIG_XML_LOAD_LAX
: Lax loading. Ignore unknown namespaces, elements, and attributes.

Additional flag for MAAPI_CONFIG_C and MAAPI_CONFIG_C_IOS:

MAAPI_CONFIG_AUTOCOMMIT
: A commit should be performed after each line. In this case the transaction identified by *th* is not used for the loading.

Additional flags for all CLI formats, i.e. MAAPI_CONFIG_J, MAAPI_CONFIG_C, and MAAPI_CONFIG_C_IOS:

MAAPI_CONFIG_CONTINUE_ON_ERROR
: Do not abort the load when an error is encountered.

MAAPI_CONFIG_SUPPRESS_ERRORS
: Do not display the long error message but instead a oneline error with the line number.

The other *flags* parameters are the same as for maapi_save_config().

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE, CONFD_ERR_BADPATH, CONFD_ERR_BAD_CONFIG, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_PROTOUSAGE, CONFD_ERR_EXTERNAL

```
int maapi_load_config_cmds(int sock, int thandle, int flags, const char
*cmds, const char *fmt, ...);
```

This function loads a configuration like maapi_load_config(), but reads the configuration from the string *cmds* instead of from a file. The *th* and *flags* parameters are the same as for maapi_load_config().

An optional *chroot* path can be given. This is only used with the MAAPI_CONFIG_C flag set.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE, CONFD_ERR_BADPATH, CONFD_ERR_BAD_CONFIG, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_PROTOUSAGE, CONFD_ERR_EXTERNAL

`int` **`maapi_load_config_stream`**`(int sock, int thandle, int flags);`

This function loads a configuration like `maapi_load_config()`, but reads the configuration from a ConfD stream socket instead of from a file. The *`th`* and *`flags`* parameters are the same as for `maapi_load_config()`.

The function returns `CONFD_ERR` on error or a positive integer id that can subsequently be used together with `confd_stream_connect()`. ConfD will read all data from the stream socket until it receives EOF. Thus the following code snippet indicates the usage pattern of this function.

```
int id;
int streamsock;
struct sockaddr_in addr;

id = maapi_load_config_stream(sock, th, flags);
if (id < 0) {
    ... handle error ...
}

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

streamsock = socket(PF_INET, SOCK_STREAM, 0);
confd_stream_connect(streamsock, (struct sockaddr*)&addr,
                     sizeof(struct sockaddr_in), id, 0);
```

Once the stream socket is connected we can write the configuration data on the socket. When we have written the complete configuration, we must close the socket, to make ConfD receive EOF. To check if the configuration load was successful we use the function `maapi_load_config_stream_result()`.

The stream socket must be connected within 10 seconds after the id is received.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE, CONFD_ERR_PROTOUSAGE, CONFD_ERR_EXTERNAL

`int` **`maapi_load_config_stream_result`**`(int sock, int id);`

We use this function to verify that the configuration we wrote on the stream socket was successfully loaded. The *`sock`* parameter must be the same maapi socket we used for `maapi_load_config_stream()` and the *`id`* parameter is the *`id`* returned by `maapi_load_config_stream()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADTYPE, CONFD_ERR_BADPATH, CONFD_ERR_BAD_CONFIG, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

`int` **`maapi_roll_config`**`(int sock, int thandle, const char *fmtpath, ...);`

This function can be used to save the equivalent of a rollback file for a given configuration before it is committed (or a subtree thereof) in curly bracket format.

The provided path indicates where we want the configuration to be rooted. It must be a prefix prepended keypath. If *`fmtpath`* is NULL, a rollback config for the entire configuration is dumped. If for example *`fmtpath`* is `"/aaa:aaa/authentication/users"` we create a rollback config for a part of the AAA data. It is not possible to extract non-config data using this function.

The function returns `CONFD_ERR` on error or a positive integer id that can subsequently be used together with `confd_stream_connect()`. Thus this function doesn't save the rollback configuration to a file,

but rather it returns an integer that is used together with a ConfD stream socket. ConfD will write all data in a stream on that socket and when done, ConfD will close its end of the socket. Thus the following code snippet indicates the usage pattern of this function.

```
int id;
int streamsock;
struct sockaddr_in addr;

id = maapi_roll_config(sock, tid, path);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET;
addr.sin_port = htons(CONFD_PORT);

streamsock = socket(PF_INET, SOCK_STREAM, 0);
confd_stream_connect(streamsock, (struct sockaddr*)&addr,
                     sizeof (struct sockaddr_in), id,0);
```

Once the stream socket is connected we can read the configuration data on the socket. We need to continue reading until we receive EOF on the socket. To check if the configuration retrieval was successful we use the function `maapi_roll_config_result()`.

The stream socket must be connected within 10 seconds after the id is received.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BAD_TYPE

`int` **`maapi_roll_config_result`**`(int` *`sock`*`, int` *`id`*`);`

We use this function to assert that we received the entire rollback configuration over a stream socket. The *`sock`* parameter must be the same maapi socket we used for `maapi_roll_config()` and the *`id`* parameter is the *`id`* returned by `maapi_roll_config()`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_ACCESS_DENIED, CONFD_ERR_EXTERNAL

`int` **`maapi_get_stream_progress`**`(int` *`sock`*`, int` *`id`*`);`

In some cases (e.g. an action or custom command that can be interrupted by the user) it may be useful to be able to terminate ConfD's reading of data from a stream socket (by closing the socket) without waiting for a potentially large amount of data written to the socket to be consumed by ConfD. This function allows us to limit the amount of data "in flight" between the application and ConfD, by reporting the amount of data read by ConfD so far.

The *`sock`* parameter must be the maapi socket used for a function call that required a stream socket for writing to ConfD (currently the only such function is `maapi_load_config_stream()`), and the *`id`* parameter is the *`id`* returned by that function. `maapi_get_stream_progress()` returns the number of bytes that ConfD has read from the stream socket. If *`id`* does not identify a stream socket that is currently being read by ConfD, the function returns CONFD_ERR with `confd_errno` set to CONFD_ERR_NOEXISTS. This can be due to e.g. that the socket has been closed, or that an error has occurred - but also that ConfD has determined that all the data has been read (e.g. the end of an XML document has been read). To avoid the latter case, the function should only be called when we have more data to write, and before the writing of that data. The following code shows a possible way to use this function.

```
#define MAX_IN_FLIGHT 4096

char buf[BUFSIZ];
```

```
int sock, streamsock, id;
int n, n_written = 0, n_read = 0;
int result;
...

while (!do_abort() && (n = get_data(buf, sizeof(buf))) > 0) {
    while (n_written - n_read > MAX_IN_FLIGHT) {
        if ((n_read = maapi_get_stream_progress(sock, id)) < 0) {
            ... handle error ...
        }
    }
    if (write(streamsock, buf, n) != n) {
        ... handle error ...
    }
    n_written += n;
}
close(streamsock);
result = maapi_load_config_stream_result(sock, id);
```

Note

A call to `maapi_get_stream_progress()` does not return until the number of bytes read has increased from the previous call (or if there is an error). This means that the above code does not imply busy-looping, but also that if the code was to call `maapi_get_stream_progress()` when `n_read` == `n_written`, the result would be a deadlock.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_NOEXISTS

`int` **`maapi_xpath_eval`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`expr`*`, int (*`*`result`*`)(confd_hkeypath_t *`*`kp`*`, confd_value_t *`*`v`*`, void *`*`state`*`), void (*`*`trace`*`)(char *), void *`*`initstate`*`, const char *`*`fmtpath`*`, ...);`

This function evaluates the XPath Path expression as supplied in *`expr`*. For each node in the resulting node set the function *`result`* is called with the keypath to the resulting node as the first argument, and, if the node is a leaf and has a value, the value of that node as the second argument. The expression will be evaluated using the root node as the context node, unless a path to an existing node is given as the last argument. For each invocation the `result()` function should return `ITER_CONTINUE` to tell the XPath evaluator to continue with the next resulting node. To stop the evaluation the `result()` can return `ITER_STOP` instead.

The *`trace`* is a pointer to a function that takes a single string as argument. If supplied it will be invoked when the xpath implementation has trace output for the current expression. (For an easy start, for example the `puts(3)` will print the trace output to stdout). If no trace is wanted `NULL` can be given.

The *`initstate`* parameter can be used for any user supplied opaque data (i.e. whatever is supplied as *`initstate`* is passed as *`state`* to the `result()` function for each invocation).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_XPATH

`int` **`maapi_xpath_eval_expr`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`expr`*`, char **`*`res`*`, void (*`*`trace`*`)(char *), const char *`*`fmtpath`*`, ...);`

Evaluate the XPath expression given in *`expr`* and return the result as a string, pointed to by *`res`*. If the call succeeds, *`res`* will point to a malloc:ed string that the caller needs to free. If the call fails *`res`* will be set to `NULL`.

It is possible to supply a path which will be treated as the initial context node when evaluating *expr* (i.e. if the path is relative, this is treated as the starting point, and this is also the node that `current()` will return when used in the XPath expression). If NULL is given, the current maapi position is used.

The *trace* is a pointer to a function that takes a single string as argument. If supplied it will be invoked when the xpath implementation has trace output for the current expression. (For an easy start, for example the `puts(3)` will print the trace output to stdout). If no trace is wanted `NULL` can be given.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH, CONFD_ERR_XPATH

`int` **`maapi_query_start`**`(int` *`sock`*`, int` *`thandle`*`, const char *`*`expr`*`, const char *`*`context_node`*`, int` *`chunk_size`*`, int` *`initial_offset`*`, enum confd_query_result_type` *`result_as`*`, int` *`nselect`*`, const char *`*`select[]`*`, int` *`nsort`*`, const char *`*`sort[]`*`);`

Start a new query attached to the transaction given in *th*. If successful a query handle is returned (the query handle is then used in subsequent calls to `maapi_query_result()` etc). Brief summary of all parameters:

sock	A previously opened maapi socket.
th	A transaction handle to a previously started transaction.
expr	The primary XPath expression.
context_node	The context node (an ikeypath) for the primary expression. `NULL` is legal, and means that the context node will be `/`.
chunk_size	How many results to return at a time. If set to 0 a default number will be used.
initial_offset	Which result in line to begin with (1 means to start from the begining).
result_as	The format the results will be returned in.
nselect	The number of expressions in the *select* parameter.
select	An array of XPath "select" expressions, of length *nselect*.
nsort	The number of expressions in the *sort* parameter.
sort	An array of XPath expressions which will be used for sorting, of length *nselect*.

A query is a way of evaluating an XPath expression and returning the results in chunks. The usage pattern is as follows: a primary expression in provided in the *expr* argument, which must evaluate to a node-set, the "results". For each node in the results node-set every "select" expression is evaluated with the result node as its context node. For example, given the YANG snippet:

```
list interface {
  key name;
  unique number;
  leaf name {
    type string;
  }
  leaf number {
    type uint32;
```

```
    mandatory true;
  }
  leaf enabled {
    type boolean;
    default true;
  }
...
}
```

and given that we want to find the name and number of all enabled interfaces - the *expr* could be `"/interface[enabled='true']"`, and the select expressions would be `{ "name", "number" }`. Note that the select expressions can have any valid XPath expression, so if you wanted to find out an interfaces name, and whether its number is even or not, the expressions would be: `{ "name", "(number mod 2) == 0" }`.

The results are then fetched using the `maapi_query_result()` function, which returns the results on the format specified by the *result_as* parameter. There are four different types of result, as defined by the type enum confd_query_result_type:

```
enum confd_query_result_type {
    CONFD_QUERY_STRING = 0,
    CONFD_QUERY_HKEYPATH = 1,
    CONFD_QUERY_HKEYPATH_VALUE = 2,
    CONFD_QUERY_TAG_VALUE = 3
};
```

I.e. the results can be returned as strings, hkeypaths, hkeypaths and values, or tags and values. The string is just the resulting string of evaluating the select XPath expression. For hkeypaths, tags, and values it is the path/tag/value of the *node that the select XPath expression evaluates to*. This means that care must be taken so that the combination of select expression and return types actually yield sensible results (for example "1 + 2" is a valid select XPath expression, and would result in the string "3" when setting the result type to `CONFD_QUERY_STRING` - but it is not a node, and thus have no hkeypath, tag, or value). A complete example:

```
qh = maapi_query_start(s, th, "/interface[enabled='true']", NULL,
                       1000, 1, CONFD_QUERY_TAG_VALUE,
                       2, (char *[]){ "name", "number" }, 0, NULL);
n = 0;
do {
  maapi_query_result(s, qh, &qr);
  n = qr->nresults;
  for (i=0; i<n; i++) {
    printf("result %d:\n", i + qr->offset);
    for (j=0; j<qr->nelements; j++) {
      // We know the type is tag-value
      char *tag = confd_hash2str(qr->results[i].tv[j].tag.tag);
      confd_pp_value(tmpbuf, BUFSIZ, &qr->results[i].tv[j].v);
      printf("  %s: %s\n", tag, tmpbuf);
    }
  }
  maapi_query_free_result(qr);
} while (n > 0);
maapi_query_stop(s, qh);
```

It is possible to sort the results using the built-in XPath function `sort-by()` (see the tailf_yang_extensions(5) man page)

It is also possible to sort the result using any expressions passed in the *sort* array. These array will be used to construct a temporary index which will live as long as the query is active. For example to start a query sorting first on the enabled leaf, and then on number one would call:

```
qh = maapi_query_start(s, th, "/interface[enabled='true']", NULL,
                       1000, 1, CONFD_QUERY_TAG_VALUE,
                       3, (char *[]){ "name", "number", "enabled" },
                       2, (char *[]){ "enabled", "number" });
...
```

Note that the index the query constructs is kept in memory, which will be released when the query is stopped.

`int` **`maapi_query_result`**`(int` *`sock`*`, int` *`qh`*`, struct confd_query_result **`qrs`*`);`

Fetch the next available chunk of results associated with query handle *qh*. The results are returned in a struct confd_query_result, which is allocated by the library. The structure is defined as:

```
struct confd_query_result {
    enum confd_query_result_type type;
    int offset;
    int nresults;
    int nelements;
    union {
        char **str;
        confd_hkeypath_t *hkp;
        struct {
            confd_hkeypath_t hkp;
            confd_value_t    val;
        } *kv;
        confd_tag_value_t *tv;
    } *results;
    void *__internal;           /* confd_lib internal housekeeping */
};
```

The *type* will always be the same as was requested in the call to `maapi_query_start()`, it is there to indicate which of the pointers in the union to use. The *offset* is the number of the first result in this chunk (i.e. for the first chunk it will be 1). How many results that are in this chunk is indicated in *nresults*, when there are no more available results it will be set to 0. Each result consists of *nelements* elements (this number is the same as the number of select parameters given in the call to `maapi_query_start()`.

All data pointed to in the result struct (as well as the struct itself) is allocated by the library - and when finished processing the result the user must call `maapi_query_free_result()` to free this data.

`int` **`maapi_query_free_result`**`(struct confd_query_result *`*`qrs`*`);`

The struct confd_query_result returned by `maapi_query_result()` is dynamically allocated (and it also contains pointers to other dynamically allocated data) and so it needs to be freed when the result has been processed. Use this function to free the struct confd_query_result (and its accompanying data) returned by `maapi_query_result()`.

`int` **`maapi_query_reset`**`(int` *`sock`*`, int` *`qh`*`);`

Reset / rewind a running query so that it starts from the beginning again. Next call to `maapi_query_result()` will then return the first chunk of results. The function can be called at any

time (i.e. both after all results have been returned to essentially run the same query again, as well as after fetching just one or a couple of results).

```
int maapi_query_reset_to(int sock, int qh, int offset);
```

Like `maapi_query_reset()`, except after the query has been reset it is restarted with the initial offset set to *`offset`*. Next call to `maapi_query_result()` will then return the first chunk of results at that offset. The function can be called at any time (i.e. both after all results have been returned to essentially run the same query again, as well as after fetching just one or a couple of results).

```
int maapi_query_stop(int sock, int qh);
```

Stops the running query identified by *`qh`*, and makes ConfD free up any internal resources associated with the query. If a query isn't explicitly closed using this call it will be cleaned up when the transaction the query is linked to ends.

```
int maapi_install_crypto_keys(int sock);
```

It is possible to define DES3 and AES keys inside confd.conf. These keys are used by ConfD to encrypt data which is entered into the system which has either of the two builtin types tailf:des3-cbc-encrypted-string or tailf:aes-cfb-128-encrypted-string. See confd_types(3).

This function will copy those keys from ConfD (which reads confd.conf) into memory in the library. To decrypt data of these types, use the function `confd_decrypt()`, see confd_lib_lib(3).

```
int maapi_do_display(int sock, int thandle, const char *fmtpath, ...);
```

If the data model uses the YANG `when` or `tailf:display-when` statement, this function can be used to determine if the item given by *`fmtpath, ...`* should be displayed or not.

```
int maapi_init_upgrade(int sock, int timeoutsecs, int flags);
```

This is the first of three functions that must be called in sequence to perform an in-service data model upgrade, i.e. replace fxs files etc without restarting the ConfD daemon. See the In-service Data Model Upgrade chapter in the User Guide for a detailed description of this procedure.

This function initializes the upgrade procedure. The *`timeoutsecs`* parameter specifies a maximum time to wait for users to voluntarily exit from "configure mode" sessions in CLI and Web UI. If transactions are still active when the timeout expires, the function will by default fail with CONFD_ERR_TIMEOUT. If the flag MAAPI_UPGRADE_KILL_ON_TIMEOUT was given via the *`flags`* parameter, such transactions will instead be forcibly terminated, allowing the initialization to complete successfully.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_LOCKED, CONFD_ERR_BADSTATE, CONFD_ERR_HA_WITH_UPGRADE, CONFD_ERR_TIMEOUT, CONFD_ERR_ABORTED

```
int maapi_perform_upgrade(int sock, const char **loadpathdirs, int n);
```

When `maapi_init_upgrade()` has completed successfully, this function must be called to instruct ConfD to load the new data model files. The *`loadpathdirs`* parameter is an array of *`n`* strings that specify the directories to load from, corresponding to the `/confdConfig/loadPath/dir` elements in `confd.conf` (see confd.conf(5)).

These directories will also be searched for CDB "init files" (see the CDB chapter in the User Guide). I.e. if the upgrade needs such files, we can place them in one of the new load path directories - or we can include directories that are used *only* for CDB "init files" in the *`loadpathdirs`* array, corresponding to the `/confdConfig/cdb/initPath/dir` elements that can be specified in `confd.conf`.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADSTATE, CONFD_ERR_BAD_CONFIG

```
int maapi_commit_upgrade(int sock);
```

When also `maapi_perform_upgrade()` has completed successfully, this function must be called to make the upgrade permanent. This includes committing the CDB upgrade transaction when CDB is used, and we can thus get all the different validation errors that can otherwise result from `maapi_apply_trans()`.

When `maapi_commit_upgrade()` has completed successfully, the program driving the upgrade must also make sure that the `/confdConfig/loadPath/dir` elements in `confd.conf` reference the new directories. If CDB "init files" are used in the upgrade as described for `maapi_commit_upgrade()` above, the program should also make sure that the `/confdConfig/cdb/initPath/dir` elements reference the directories where those files are located.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADSTATE, CONFD_ERR_NOTSET, CONFD_ERR_NON_UNIQUE, CONFD_ERR_BAD_KEYREF, CONFD_ERR_TOO_FEW_ELEMS, CONFD_ERR_TOO_MANY_ELEMS, CONFD_ERR_UNSET_CHOICE, CONFD_ERR_MUST_FAILED, CONFD_ERR_MISSING_INSTANCE, CONFD_ERR_INVALID_INSTANCE, CONFD_ERR_BADTYPE, CONFD_ERR_EXTERNAL

```
int maapi_abort_upgrade(int sock);
```

Calling this function at any point before the call of `maapi_commit_upgrade()` will abort the upgrade.

Note

`maapi_abort_upgrade()` should *not* be called if any of the three previous functions fail - in that case, ConfD will do an internal abort of the upgrade.

CONFD DAEMON CONTROL

```
int maapi_aaa_reload(int sock, int synchronous);
```

When the ConfD AAA tree is populated by an external data provider (see the AAA chapter in the User Guide), this function can be used by the data provider to notify ConfD when there is a change to the AAA data. I.e. it is an alternative to executing the command **confd --clear-aaa-cache**.

If the *synchronous* parameter is 0, the function will only initiate the loading of the AAA data, just like **confd --clear-aaa-cache** does, and return CONFD_OK as long as the communication with ConfD succeeded. Otherwise it will wait for the loading to complete, and return CONFD_OK only if the loading was successful.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_EXTERNAL

```
int maapi_aaa_reload_path(int sock, int synchronous, const char *fmt, ...);
```

A variant of `maapi_aaa_reload()` that causes only the AAA subtree given by the path in *fmt* to be loaded. This may be useful to load changes to the AAA data when loading the complete AAA tree from an external data provider takes a long time. Obviously care must be taken to make sure that all changes actually get loaded, and a complete load using e.g. `maapi_aaa_reload()` should be done at least when ConfD is started. The path may specify a container or list entry, but not a specific leaf.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_EXTERNAL

`int` **`maapi_start_phase`**`(int` *`sock`*`, int` *`phase`*`, int` *`synchronous`*`);`

Once the ConfD daemon has been started in phase0 it is possible to use this function to tell the daemon to proceed to startphase 1 or 2 (as indicated in the *`phase`* parameter). If *`synchronous`* is non-zero the call does not return until the daemon has completed the transition to the requested start phase.

Note that start-phase1 can fail, (see documentation of `--start-phase1` in confd(1)) in particular if CDB fails. In that case `maapi_start_phase()` will return CONFD_ERR, with confderrno set to CONFD_ERR_START_FAILED. However if ConfD stops before it has a chance to send back the error CONFD_EOF might be returned.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_START_FAILED

`int` **`maapi_wait_start`**`(int` *`sock`*`, int` *`phase`*`);`

To synchronize startup with ConfD this function can be used to wait for ConfD to reach a particular start phase (0, 1, or 2). Note that to implement an equivalent of **confd --wait-started** or **confd --wait-phase0** case must also be taken to retry `maapi_connect()`, which will fail until ConfD has started enough to accept connections to its IPC port.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_PROTOUSAGE

`int` **`maapi_stop`**`(int` *`sock`*`, int` *`synchronous`*`);`

Request the ConfD daemon to stop, if *`synchronous`* is non-zero the call will wait until ConfD has come to a complete halt. Note that since the daemon exits, the socket won't be re-usable after this call. Equivalent to **confd --stop**.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_reload_config`**`(int` *`sock`*`);`

Request that the ConfD daemon reloads its configuration files. The daemon will also close and re-open its log files. Equivalent to **confd --reload**.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_reopen_logs`**`(int` *`sock`*`);`

Request that the ConfD daemon closes and re-opens its log files, useful for logrotate(8).

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS

`int` **`maapi_rebind_listener`**`(int` *`sock`*`, int` *`listener`*`);`

Request that the subsystem(s) specified by *`listener`* rebinds its listener socket(s). Currently open sockets (if any) will be closed, and new sockets created and bound via `bind(2)` and `listen(2)`. This is useful e.g. if `/confdConfig/ignoreBindErrors/enabled` is set to "true" in `confd.conf`, and some bindings have failed due to a problem that subsequently has been fixed. Calling this function then avoids the disable/enable config change that would otherwise be required to cause a rebind.

The following values can be used for the *`listener`* parameter, ORed together if more than one:

```
#define CONFD_LISTENER_IPC     (1 << 0)
#define CONFD_LISTENER_NETCONF (1 << 1)
#define CONFD_LISTENER_SNMP    (1 << 2)
```

```
#define CONFD_LISTENER_CLI     (1 << 3)
#define CONFD_LISTENER_WEBUI   (1 << 4)
```

Note

It is not possible to rebind sockets for northbound listeners during the transition from start phase 1 to start phase 2. If this is attempted, the call will fail (and do nothing) with `confd_errno` set to CONFD_ERR_BADSTATE.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADSTATE

`int` **`maapi_clear_opcache`**`(int` *`sock`*`, const char *`*`fmt`*`, ...);`

Request clearing of the operational data cache (see the Operational Data chapter in the User Guide). A path can be given via the *`fmt`* and subsequent parameters, to clear only the cached data for the subtree designated by that path. To clear the whole cache, pass NULL or "/" for *`fmt`*.

Errors: CONFD_ERR_MALLOC, CONFD_ERR_OS, CONFD_ERR_BADPATH

SEE ALSO

confd_lib(3) - Confd lib

confd_types(3) - ConfD C data types

The ConfD User Guide

Name

confd_types — ConfD value representation in C

Synopsis

#include <confd_lib.h>

DESCRIPTION

The `libconfd` library manages data values such as elements received over the NETCONF protocol. This man page describes how these values as well as the XML paths (confd_hkeypath_t) identifying the values are represented in the C language.

TYPEDEFS

The following `enum` defines the different types. These are used to represent data model types from several different sources - see the section DATA MODEL TYPES at the end of this manual page for a full specification of how the data model types map to these types.

```
enum confd_vtype {
    C_NOEXISTS    = 1,  /* end marker                              */
    C_XMLTAG      = 2,  /* struct xml_tag                         */
    C_SYMBOL      = 3,  /* not yet used                           */
    C_STR         = 4,  /* NUL-terminated strings                 */
    C_BUF         = 5,  /* confd_buf_t (string ...)               */
    C_INT8        = 6,  /* int8_t    (int8)                       */
    C_INT16       = 7,  /* int16_t   (int16)                      */
    C_INT32       = 8,  /* int32_t   (int32)                      */
    C_INT64       = 9,  /* int64_t   (int64)                      */
    C_UINT8       = 10, /* u_int8_t  (uint8)                      */
    C_UINT16      = 11, /* u_int16_t (uint16)                     */
    C_UINT32      = 12, /* u_int32_t (uint32)                     */
    C_UINT64      = 13, /* u_int64_t (uint64)                     */
    C_DOUBLE      = 14, /* double (xs:float,xs:double)            */
    C_IPV4        = 15, /* struct in_addr in NBO                  */
                        /*  (inet:ipv4-address)                   */
    C_IPV6        = 16, /* struct in6_addr in NBO                 */
                        /*  (inet:ipv6-address)                   */
    C_BOOL        = 17, /* int       (boolean)                    */
    C_QNAME       = 18, /* struct confd_qname (xs:QName)          */
    C_DATETIME    = 19, /* struct confd_datetime                  */
                        /*  (yang:date-and-time)                  */
    C_DATE        = 20, /* struct confd_date (xs:date)            */
    C_TIME        = 23, /* struct confd_time (xs:time)            */
    C_DURATION    = 27, /* struct confd_duration (xs:duration)    */
    C_ENUM_VALUE  = 28, /* int32_t (enumeration)                  */
    C_BIT32       = 29, /* u_int32_t (bits size 32)               */
    C_BIT64       = 30, /* u_int64_t (bits size 64)               */
    C_LIST        = 31, /* confd_list (leaf-list)                 */
    C_XMLBEGIN    = 32, /* struct xml_tag, start of container or  */
                        /*  list entry                            */
    C_XMLEND      = 33, /* struct xml_tag, end of container or    */
                        /*  list entry                            */
    C_OBJECTREF   = 34, /* struct confd_hkeypath*                 */
                        /*  (instance-identifier)                 */
    C_UNION       = 35, /* (union) - not used in API functions    */
    C_PTR         = 36, /* see cdb_get_values in confd_lib_cdb(3) */
```

```
    C_CDBBEGIN    = 37, /* as C_XMLBEGIN, with CDB instance index  */
    C_OID         = 38, /* struct confd_snmp_oid*                  */
                        /*  (yang:object-identifier)              */
    C_BINARY      = 39, /* confd_buf_t (binary ...)                */
    C_IPV4PREFIX  = 40, /* struct confd_ipv4_prefix                */
                        /*  (inet:ipv4-prefix)                    */
    C_IPV6PREFIX  = 41, /* struct confd_ipv6_prefix                */
                        /*  (inet:ipv6-prefix)                    */
    C_DEFAULT     = 42, /* default value indicator                 */
    C_DECIMAL64   = 43, /* struct confd_decimal64 (decimal64)      */
    C_IDENTITYREF = 44, /* struct confd_identityref (identityref)  */
    C_XMLBEGINDEL = 45, /* as C_XMLBEGIN, but for a deleted list   */
                        /*  entry                                 */
    C_DQUAD       = 46, /* struct confd_dotted_quad                */
                        /*  (yang:dotted-quad)                    */
    C_HEXSTR      = 47, /* confd_buf_t (yang:hex-string)           */
    C_IPV4_AND_PLEN = 48, /* struct confd_ipv4_prefix              */
                        /*  (tailf:ipv4-address-and-prefix-length) */
    C_IPV6_AND_PLEN = 49, /* struct confd_ipv6_prefix              */
                        /*  (tailf:ipv6-address-and-prefix-length) */
    C_MAXTYPE           /* maximum marker; add new values above    */
};
```

A concrete value is represented as a confd_value_t C struct:

```
typedef struct confd_value {
    enum confd_vtype type;  /* as defined above */
    union {
        struct xml_tag xmltag;
        u_int32_t symbol;
        confd_buf_t buf;
        confd_buf_const_t c_buf;
        char *s;
        const char *c_s;
        int8_t i8;
        int16_t i16;
        int32_t i32;
        int64_t i64;
        u_int8_t u8;
        u_int16_t u16;
        u_int32_t u32;
        u_int64_t u64;
        double d;
        struct in_addr ip;
        struct in6_addr ip6;
        int boolean;
        struct confd_qname qname;
        struct confd_datetime datetime;
        struct confd_date date;
        struct confd_time time;
        struct confd_duration duration;
        int32_t enumvalue;
        u_int32_t b32;
        u_int64_t b64;
        struct confd_list list;
        struct confd_hkeypath *hkp;
        struct confd_vptr ptr;
        struct confd_snmp_oid *oidp;
        struct confd_ipv4_prefix ipv4prefix;
        struct confd_ipv6_prefix ipv6prefix;
```

```
        struct confd_decimal64 d64;
        struct confd_identityref idref;
        struct confd_dotted_quad dquad;
        u_int32_t enumhash;     /* backwards compat */
    } val;
} confd_value_t;
```

C_NOEXISTS — This is used internally by ConfD, as an end marker in confd_hkeypath_t arrays, and as a "value does not exist" indicator in arrays of values.

C_DEFAULT — This is used to indicate that an element with a default value defined in the data model does not have a value set. When reading data from ConfD, we will only get this indication if we specifically request it, otherwise the default value is returned.

C_XMLTAG — An C_XMLTAG value is represented as a struct:

```
struct xml_tag {
    u_int32_t tag;
    u_int32_t ns;
};
```

When a YANG module is compiled by the confdc(1) compiler, the `--emit-h` flag is used to generate a .h file containing definitions for all the nodes in the module. For example if we compile the following YANG module:

```
# cat blaster.yang
module blaster {
  namespace "http://tail-f.com/ns/blaster";
  prefix blaster;

  import tailf-common {
    prefix tailf;
  }

  typedef Fruit {
    type enumeration {
      enum apple;
      enum orange;
      enum pear;
    }
  }
  container tiny {
    tailf:callpoint xcp;
    leaf foo {
      type int8;
    }
    leaf bad {
      type int16;
    }
  }
}

# confdc -c blaster.yang
# confdc --emit-h blaster.h blaster.fxs
```

We get the following contents in blaster.h

```
# cat blaster.h
/*
 * BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE
```

```
 * This file has been auto-generated by the confdc compiler.
 * Source: blaster.fxs
 * BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE BEWARE
 */

#ifndef _BLASTER_H_
#define _BLASTER_H_

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#ifndef blaster__ns
#define blaster__ns 670579579
#define blaster__ns_id "http://tail-f.com/ns/blaster"
#define blaster__ns_uri "http://tail-f.com/ns/blaster"
#endif

#define blaster_orange 1
#define blaster_apple 0
#define blaster_pear 2
#define blaster_foo 161968632
#define blaster_tiny 1046642021
#define blaster_bad 1265139696
#define blaster__callpointid_xcp "xcp"

#ifdef __cplusplus
}
#endif

#endif
```

The integers in the .h file are used in the struct xml_tag, thus the container node `tiny` is represented as a xml_tag C struct `{tag=1046642021, ns=670579579}` or, using the #defines `{tag=blaster_tiny, ns=blaster__ns}`.

Each callpoint, actionpoint, and validate statement also yields a preprocessor symbol. If the symbol is used rather than the literal string in calls to ConfD, the C compiler will catch the potential problem when the id in the data model has changed but the C code hasn't been updated.

Sometimes we wish to retrieve a string representation of defined hash values. This can be done with the function `confd_hash2str()`, see the USING SCHEMA INFORMATION section below.

`C_BUF` This type is used to represent the YANG built-in type string and the xs:token type. The struct which is used is:

```
typedef struct confd_buf {
    unsigned int size;
    unsigned char *ptr;
} confd_buf_t;
```

Strings passed to the application from ConfD are always NUL-terminated. When values of this type are received by the callback functions in confd_lib_dp(3), the ptr field is a pointer to libconfd private memory, and the data will not survive unless copied by the application.

To create and extract values of type C_BUF we do:

```
confd_value_t myval;
char *x; int len;

CONFD_SET_BUF(&myval, "foo", 3)
x = CONFD_GET_BUFPTR(&myval);
len = CONFD_GET_BUFSIZE(&myval);
```

It is important to realize that C_BUF data received by the application through either `maapi_get_elem()` or `cdb_get()` which are of type C_BUF must be freed by the application.

C_STR

This tag is never received by the application. Values and keys received in the various data callbacks (See `confd_register_data_cb()` in confd_lib_dp(3) never have this type. It is only used when the application replies with values to ConfD. (See `confd_data_reply_value()` in confd_lib_dp(3)).

It is used to represent regular NUL-terminated char* values. Example:

```
confd_value_t myval;
myval.type = C_STR;
myval.val.s = "Zaphod";
/* or alternatively and recommended */
CONFD_SET_STR(&myval, "Beeblebrox");
```

C_INT8

Used to represent the YANG built-in type int8, which is a signed 8 bit integer. The corresponding C type is int8_t. Example:

```
int8_t ival;
confd_value_t myval;

CONFD_SET_INT8(&myval, -32);
ival = CONFD_GET_INT8(&myval);
```

C_INT16

Used to represent the YANG built-in type int16, which is a signed 16 bit integer. The corresponding C type is int16_t. Example:

```
int16_t ival;
confd_value_t myval;

CONFD_SET_INT16(&myval, -3277);
ival = CONFD_GET_INT16(&myval);
```

C_INT32

Used to represent the YANG built-in type int32, which is a signed 32 bit integer. The corresponding C type is int32_t. Example:

```
int32_t ival;
confd_value_t myval;

CONFD_SET_INT32(&myval, -77732);
ival = CONFD_GET_INT32(&myval);
```

C_INT64

Used to represent the YANG built-in type int64, which is a signed 64 bit integer. The corresponding C type is int64_t. Example:

```
int64_t ival;
```

```
confd_value_t myval;

CONFD_SET_INT64(&myval, -32);
ival = CONFD_GET_INT64(&myval);
```

C_UINT8 — Used to represent the YANG built-in type uint8, which is an unsigned 8 bit integer. The corresponding C type is u_int8_t. Example:

```
u_int8_t ival;
confd_value_t myval;

CONFD_SET_UINT8(&myval, 32);
ival = CONFD_GET_UINT8(&myval);
```

C_UINT16 — Used to represent the YANG built-in type uint16, which is an unsigned 16 bit integer. The corresponding C type is u_int16_t. Example:

```
u_int16_t ival;
confd_value_t myval;

CONFD_SET_UINT16(&myval, 3277);
ival = CONFD_GET_UINT16(&myval);
```

C_UINT32 — Used to represent the YANG built-in type uint32, which is an unsigned 32 bit integer. The corresponding C type is u_int32_t. Example:

```
u_int32_t ival;
confd_value_t myval;

CONFD_SET_UINT32(&myval, 77732);
ival = CONFD_GET_UINT32(&myval);
```

C_UINT64 — Used to represent the YANG built-in type uint64, which is an unsigned 64 bit integer. The corresponding C type is u_int64_t. Example:

```
u_int64_t ival;
confd_value_t myval;

CONFD_SET_UINT64(&myval, 32);
ival = CONFD_GET_UINT64(&myval);
```

C_DOUBLE — Used to represent the XML schema types xs:decimal, xs:float and xs:double. They are all coerced into the C type double. Example:

```
double d;
confd_value_t myval;

CONFD_SET_DOUBLE(&myval, 3.14);
d = CONFD_GET_DOUBLE(&myval);
```

C_BOOL — Used to represent the YANG built-in type boolean. The C representation is an integer with `0` representing false and non-zero representing true. Example:

```
int bool
confd_value_t myval;

CONFD_SET_BOOL(&myval, 1);
b = CONFD_GET_BOOL(&myval);
```

`C_QNAME` Used to represent XML Schema type xs:QName which consists of a pair of strings, prefix and a name. Data is allocated by the library as for C_BUF. Example:

```
unsigned char* prefix, *name;
int prefix_len, name_len;
confd_value_t myval;

CONFD_SET_QNAME(&myval, "myprefix", 8, "myname", 6);
prefix = CONFD_GET_QNAME_PREFIX_PTR(&myval);
prefix_len = CONFD_GET_QNAME_PREFIX_SIZE(&myval);
name = CONFD_GET_QNAME_NAME_PTR(&myval);
name_len = CONFD_GET_QNAME_NAME_SIZE(&myval);
```

`C_DATETIME` Used to represent the YANG type yang:date-and-time. The C representation is a struct:

```
struct confd_datetime {
    int16_t year;
    u_int8_t month;
    u_int8_t day;
    u_int8_t hour;
    u_int8_t min;
    u_int8_t sec;
    u_int32_t micro;
    int8_t timezone;
    int8_t timezone_minutes;
};
```

ConfD does not try to convert the data values into timezone independent C structs. The timezone and timezone_minutes fields are integers where:

`timezone == 0 && timezone_minutes == 0`	represents UTC. This corresponds to a timezone specification in the string form of "Z" or "+00:00".
`-14 <= timezone && timezone <= 14`	represents an offset in hours from UTC. In this case `timezone_minutes` represents a fraction of an hour in minutes if the offset from UTC isn't an integral number of hours, otherwise it is 0. If `timezone != 0`, its sign gives the direction of the offset, and `timezone_minutes` is always `>= 0` - otherwise the sign of `timezone_minutes` gives the direction of the offset. E.g. `timezone == 5 && timezone_minutes == 30` corresponds to a timezone specification in the string form of "+05:30".
`timezone == CONFD_TIMEZONE_UNDEF`	means that the string form indicates lack of timezone information with "-00:00".

It is up to the application to transform these structs into more UNIX friendly structs such as struct tm from `<time.h>`. Example:

```
#include <time.h>
```

```
confd_value_t myval;
struct confd_datetime dt;
struct tm *tm = localtime(time(NULL));

dt.year = tm->tm_year + 1900; dt.month = tm->tm_mon + 1;
dt.day = tm->tm_mday; dt->hour = tm->tm_hour;
dt.min = tm->tm_min; dt->sec = tm->tm_sec;
dt.micro = 0; dt.timezone = CONFD_TIMEZONE_UNDEF;
CONFD_SET_DATETIME(&myval, dt);
dt = CONFD_GET_DATETIME(&myval);
```

C_DATE — Used to represent the XML Schema type xs:date. The C representation is a struct:

```
struct confd_date {
    int16_t year;
    u_int8_t month;
    u_int8_t day;
    int8_t timezone;
    int8_t timezone_minutes;
};
```

Example:

```
confd_value_t myval;
struct confd_date dt;

dt.year = 1960, dt.month = 3,
dt.day = 31; dt.timezone = CONFD_TIMEZONE_UNDEF;
CONFD_SET_DATE(&myval, dt);
dt = CONFD_GET_DATE(&myval);
```

C_TIME — Used to represent the XML Schema type xs:time. The C representation is a struct:

```
struct confd_time {
    u_int8_t hour;
    u_int8_t min;
    u_int8_t sec;
    u_int32_t micro;
    int8_t timezone;
    int8_t timezone_minutes;
};
```

Example:

```
confd_value_t myval;
struct confd_time dt;

dt.hour = 19, dt.min = 3,
dt.sec = 31; dt.timezone = CONFD_TIMEZONE_UNDEF;
CONFD_SET_TIME(&myval, dt);
dt = CONFD_GET_TIME(&myval);
```

C_DURATION — Used to represent the XML Schema type xs:duration. The C representation is a struct:

```
struct confd_duration {
    u_int32_t years;
    u_int32_t months;
    u_int32_t days;
    u_int32_t hours;
```

```
    u_int32_t mins;
    u_int32_t secs;
    u_int32_t micros;
};
```

Example of something that is supposed to last 3 seconds:

```
confd_value_t myval;
struct confd_duration dt;

memset(&dt, 0, sizeof(struct confd_duration));
dt.secs = 3;
CONFD_SET_DURATION(&myval, dt);
dt = CONFD_GET_DURATION(&myval);
```

`C_IPV4` Used to represent the YANG type inet:ipv4-address. The C representation is a struct in_addr Example:

```
struct in_addr ip;
confd_value_t myval;

ip.s_addr = inet_addr("192.168.1.2");
CONFD_SET_IPV4(&myval, ip);
ip = CONFD_GET_IPV4(&myval);
```

`C_IPV6` Used to represent the YANG type inet:ipv6-address. The C representation is as struct in6_addr Example:

```
struct in6_addr ip6;
confd_value_t myval;

inet_pton(AF_INET6, "FFFF::192.168.42.2", &ip6);
CONFD_SET_IPV6(&myval, ip6);
ip6 = CONFD_GET_IPV6(&myval);
```

`C_ENUM_VALUE` Used to represent the YANG built-in type enumeration - like the Fruit enumeration from the beginning of this man page.

```
enum fruit {
   ORANGE = blaster_orange,
   APPLE = blaster_apple,
   PEAR = blaster_pear
};

enum fruit f;
confd_value_t myval;
CONFD_SET_ENUM_VALUE(&myval, APPLE);
f = CONFD_GET_ENUM_VALUE(&myval);
```

Thus leafs that have type enumeration in the YANG module do not have values that are strings in the C code, but integer values according to the YANG standard. The file generated by **confdc --emit-h** includes `#define` symbols for these integer values.

`C_BIT32`, `C_BIT64` Used to represent the YANG built-in type bits. In C the value representation for a bitmask is either a 32 bit or a 64 bit unsigned integer, depending on the highest bit position assigned.

```
u_int32_t mask = 77;
```

```
confd_value_t myval;
CONFD_SET_BIT32(&myval, mask);
mask = CONFD_GET_BIT32(&myval);
```

C_LIST — Used to represent a YANG `leaf-list`. In C the value representation for is:

```
struct confd_list {
    unsigned int size;
    struct confd_value *ptr;
};
```

Similar to the C_BUF type, the confd library will allocate data when an element of type C_LIST is retrieved via `maapi_get_elem()` or `cdb_get()`. Using `confd_free_value()` (see confd_lib_lib(3)) to free allocated data is especially convenient for C_LIST, as the individual list elements may also have allocated data (e.g. a YANG `leaf-list` of type string).

To set a value of type C_LIST we have to populate the list array separately, for example:

```
confd_value_t arr[5];
confd_value_t v;
confd_value_t *vp;
int i, size;

for (i=0; i<5; i++)
     CONFD_SET_INT32(&arr[i], i);
CONFD_SET_LIST(&v, &arr[0], 5);

vp = CONFD_GET_LIST(&v);
size = CONFD_GET_LISTSIZE(&v);
```

C_XMLBEGIN, C_XMLEND — These are only used in the "Tagged Value Array" format for representing XML structures, see below. The representation is the same as for C_XMLTAG.

C_OBJECTREF — This is used to represent the YANG built-in type instance-identifier. Values are represented as confd_hkeypath_t pointers. Data is allocated by the library as for C_BUF. When we read an instance-identifier via e.g. `cdb_get()` we can retrieve the pointer to the keypath as:

```
confd_value_t v;
confd_hkeypath_t *hkp;

cdb_get(sock, &v, mypath);
hkp = CONFD_GET_OBJECTREF(&v);
```

To retrieve the value which is identified by the instance-identifier we can e.g. use the "%h" modifier in the format string used with the CDB and MAAPI API functions.

C_OID — This is used to represent the YANG yang:object-identifier and yang:object-identifier-128 types, i.e. SNMP Object Identifiers. The value is a pointer to a struct:

```
struct confd_snmp_oid {
    u_int32_t oid[128];
    int len;
};
```

Data is allocated by the library as for C_BUF. When using values of this type, we set or get the `len` element, and the individual OID elements in the `oid` array. This example will store the string "0.1.2" in `buf`:

```
struct confd_snmp_oid myoid;
confd_value_t myval;
char buf[BUFSIZ];
int i;

for (i = 0; i < 3; i++)
    myoid.oid[i] = i;
myoid.len = 3;
CONFD_SET_OID(&myval, &myoid);

confd_pp_value(buf, sizeof(buf), &myval);
```

`C_BINARY` This type is used to represent arbitrary binary data. The YANG built-in type binary, the ConfD built-in types tailf:hex-list and tailf:octet-list, and the XML Schema primitive type xs:hexBinary all use this type. The value representation is the same as for C_BUF. Binary (C_BINARY) data received by the application from ConfD is always NUL terminated, but since the data may also contain NUL bytes, it is generally necessary to use the size given by the representation.

```
typedef struct confd_buf {
    unsigned int size;
    unsigned char *ptr;
} confd_buf_t;
```

Data is also allocated by the library as for C_BUF. Example:

```
confd_value_t myval, myval2;
unsigned char *bin;
int len;

bin = CONFD_GET_BINARY_PTR(&myval);
len = CONFD_GET_BINARY_SIZE(&myval);
CONFD_SET_BINARY(&myval2, bin, len);
```

`C_IPV4PREFIX` Used to represent the YANG data type inet:ipv4-prefix. The C representation is a struct as follows:

```
struct confd_ipv4_prefix {
    struct in_addr ip;
    u_int8_t len;
};
```

Example:

```
struct confd_ipv4_prefix prefix;
confd_value_t myval;

prefix.ip.s_addr = inet_addr("10.0.0.0");
prefix.len = 8;
CONFD_SET_IPV4PREFIX(&myval, prefix);
prefix = CONFD_GET_IPV4PREFIX(&myval);
```

`C_IPV6PREFIX` Used to represent the YANG data type inet:ipv6-prefix. The C representation is a struct as follows:

```
struct confd_ipv6_prefix {
    struct in6_addr ip6;
    u_int8_t len;
};
```

Example:

```
struct confd_ipv6_prefix prefix;
confd_value_t myval;

inet_pton(AF_INET6, "2001:DB8::1428:57A8", &prefix.ip6);
prefix.len = 125;
CONFD_SET_IPV6PREFIX(&myval, prefix);
prefix = CONFD_GET_IPV6PREFIX(&myval);
```

`C_DECIMAL64` Used to represent the YANG built-in type decimal64, which is a decimal number with 64 bits of precision. The C representation is a struct as follows:

```
struct confd_decimal64 {
    int64_t value;
    u_int8_t fraction_digits;
};
```

The `value` element is scaled with the value of the `fraction_digits` element, to be able to represent it as a 64-bit integer. Note that `fraction_digits` is a constant for any given instance of a decimal64 type. It is provided whenever we receive a C_DECIMAL64 from ConfD. When we provide a C_DECIMAL64 to ConfD, we can set `fraction_digits` either to the correct value or to 0 - however the `value` element must always be correctly scaled. See also `confd_get_decimal64_fraction_digits()` in the confd_lib_lib(3) man page.

Example:

```
struct confd_decimal64 d64;
confd_value_t myval;

d64.value = 314159;
d64.fraction_digits = 5;
CONFD_SET_DECIMAL64(&myval, d64);
d64 = CONFD_GET_DECIMAL64(&myval);
```

`C_IDENTITYREF` Used to represent the YANG built-in type identityref, which references an existing `identity`. The C representation is a struct as follows:

```
struct confd_identityref {
    u_int32_t ns;
    u_int32_t id;
};
```

The `ns` and `id` elements are hash values that represent the namespace of the module that defines the identity, and the identity within that module.

Example:

```
struct confd_identityref idref;
confd_value_t myval;
```

```
idref.ns = des__ns;
idref.id = des_des3
CONFD_SET_IDENTITYREF(&myval, idref);
idref = CONFD_GET_IDENTITYREF(&myval);
```

C_DQUAD

Used to represent the YANG data type yang:dotted-quad. The C representation is a struct as follows:

```
struct confd_dotted_quad {
    unsigned char quad[4];
};
```

Example:

```
struct confd_dotted_quad dquad;
confd_value_t myval;

dquad.quad[0] = 1;
dquad.quad[1] = 2;
dquad.quad[2] = 3;
dquad.quad[3] = 4;
CONFD_SET_DQUAD(&myval, dquad);
dquad = CONFD_GET_DQUAD(&myval);
```

C_HEXSTR

Used to represent the YANG data type yang:hex-string. The value representation is the same as for C_BUF and C_BINARY. C_HEXSTR data received by the application from ConfD is always NUL terminated, but since the data may also contain NUL bytes, it is generally necessary to use the size given by the representation.

```
typedef struct confd_buf {
    unsigned int size;
    unsigned char *ptr;
} confd_buf_t;
```

Data is also allocated by the library as for C_BUF/C_BINARY. Example:

```
confd_value_t myval, myval2;
unsigned char *hex;
int len;

hex = CONFD_GET_HEXSTR_PTR(&myval);
len = CONFD_GET_HEXSTR_SIZE(&myval);
CONFD_SET_HEXSTR(&myval2, bin, len);
```

C_IPV4_AND_PLEN

Used to represent the ConfD built-in data type tailf:ipv4-address-and-prefix-length. The C representation is the same struct that is used for C_IPV4PREFIX, as follows:

```
struct confd_ipv4_prefix {
    struct in_addr ip;
    u_int8_t len;
};
```

Example:

```
struct confd_ipv4_prefix ip_and_len;
confd_value_t myval;
```

```
ip_and_len.ip.s_addr = inet_addr("172.16.1.2");
ip_and_len.len = 16;
CONFD_SET_IPV4_AND_PLEN(&myval, ip_and_len);
ip_and_len = CONFD_GET_IPV4_AND_PLEN(&myval);
```

C_IPV6_AND_PLEN Used to represent the ConfD built-in data type tailf:ipv6-address-and-prefix-length. The C representation is the same struct that is used for C_IPV6PREFIX, as follows:

```
struct confd_ipv6_prefix {
    struct in6_addr ip6;
    u_int8_t len;
};
```

Example:

```
struct confd_ipv6_prefix ip_and_len;
confd_value_t myval;

inet_pton(AF_INET6, "2001:DB8::1428:57A8", &ip_and_len.ip6);
ip_and_len.len = 64;
CONFD_SET_IPV6_AND_PLEN(&myval, ip_and_len);
ip_and_len = CONFD_GET_IPV6_AND_PLEN(&myval);
```

XML PATHS

Almost all of the callback functions the user is supposed write for the confd_lib_dp(3) library takes a parameter of type confd_hkeypath_t. This type includes an array of the type confd_value_t described above. The confd_hkeypath_t is defined as a C struct:

```
typedef struct confd_hkeypath {
    int len;
    confd_value_t v[MAXDEPTH][MAXKEYLEN];
} confd_hkeypath_t;
```

Where:

```
#define MAXDEPTH 20   /* max depth of data model tree
                         (max KP length + 1) */
#define MAXKEYLEN 9   /* max number of key elems
                         (max keys + 1) */
```

For example, assume we have a YANG module with:

```
container servers {
  tailf:callpoint mycp;
  list server {
    key name;
    max-elements 64;
    leaf name {
      type string;
    }
    leaf ip {
      type inet:ip-address;
    }
    leaf port {
      type inet:port-number;
    }
```

```
  }
}
```

Assuming a `server` entry with the name "www" exists, then the path `/servers/server{www}/ip` is valid and identifies the `ip` leaf in the server entry whose key is "www".

The confd_hkeypath_t which corresponds to `/servers/server{www}/ip` is received in reverse order so the following holds assuming the variable holding a pointer to the keypath is called hkp.

`hkp->v[0][0]` is the last element, the "ip" element. It is a data model node, and `CONFD_GET_XMLTAG(&hkp->v[0][0])` will evaluate to a hashed integer (which can be found in the confdc generated .h file as a #define)

`hkp->v[1][0]` is the next element in the path. The key element is called "name". This is a string value - thus `strcmp("www", CONFD_GET_BUFPTR(&hkp->v[1][0])) == 0` holds.

If we had chosen to use multiple keys in our data model - for example if we had chosen to use both the "name" and the "ip" leafs as keys:

```
key "name ip";
```

The hkeypaths would be different since two keys are required. A valid path identifying a `port` leaf would be `/servers/server{www 10.2.3.4}/port`. In this case we can get to the `ip` part of the key with:

```
struct in_addr ip;
ip = CONFD_GET_IPV4(&hkp->v[1][1])
```

USER-DEFINED TYPES

We can define new types in addition to those listed in the TYPEDEFS section above. This can be useful if none of the predefined types, nor a derivation of one of those types via standard YANG restrictions, is suitable. Of course it is always possible to define a type as a derivation of string and have the application parse the string whenever a value needs to be processed, but with a user-defined type ConfD will do the string <-> value translation just as for the predefined types.

A user-defined type will always have a value representation that uses a confd_value_t with one of the enum confd_vtype values listed above, but the textual representation and the range(s) of allowed values are defined by the user. The `misc/user_type` example in the collection delivered with the ConfD release shows implementation of several user-defined types - it will be useful to refer to it for the description below.

The choice of confd_vtype to use for the value representation can be whatever suits the actual data values best. The example uses C_INT32, C_IPV4PREFIX, and C_IPV6PREFIX for the value representation of the respective types, but in many cases the opaque byte array provided by C_BINARY will be most suitable - this can e.g. be mapped to/from an arbitrary C struct.

When we want to implement a user-defined type, we need to specify the type as string, and add a `tailf:typepoint` statement - see tailf_yang_extensions(5). We can use `tailf:typepoint` wherever a built-in or derived type can be specified, i.e. as sub-statement to `typedef`, `leaf`, or `leaf-list`:

```
typedef myType {
  type string;
  tailf:typepoint my_type;
}
```

```
container c {
  leaf one {
    type myType;
  }
  leaf two {
    type string;
    tailf:typepoint two_type;
  }
}
```

The argument to the `tailf:typepoint` statement is used to locate the type implementation, similar to how "callpoints" are used to locate data providers, but the actual mechanism is different, as described below.

To actually implement the type definition, we need to write three callback functions that are defined in the struct confd_type:

```
struct confd_type {
    /* If a derived type point at the parent */
    struct confd_type *parent;

    /* not used in confspecs, but used in YANG */
    struct confd_type *defval;

    /* parse value located in str, and validate.
     * returns CONFD_TRUE if value is syntactically correct
     * and CONFD_FALSE otherwise.
     */
    int (*str_to_val)(struct confd_type *self,
                      struct confd_type_ctx *ctx,
                      const char *str, unsigned int len,
                      confd_value_t *v);

    /* print the value to str.
     * does not print more than len bytes, including trailing NUL.
     * return value as snprintf - i.e. if the value is correct for
     * the type, it returns the length of the string form regardless
     * of the len limit - otherwise it returns a negative number.
     * thus, the NUL terminated output has been completely written
     * if and only if the returned value is nonnegative and less
     * than len.
     * If strp is non-NULL and the string form is constant (i.e.
     * C_ENUM_VALUE), a pointer to the string is stored in *strp.
     */
    int (*val_to_str)(struct confd_type *self,
                      struct confd_type_ctx *ctx,
                      const confd_value_t *v,
                      char *str, unsigned int len,
                      const char **strp);

    /* returns CONFD_TRUE if value is correct, otherwise CONFD_FALSE
     */
    int (*validate)(struct confd_type *self,
                    struct confd_type_ctx *ctx,
                    const confd_value_t *v);

    /* data optionally used by the callbacks */
    void *opaque;
};
```

I.e. `str_to_val()` and `val_to_str()` are responsible for the string to value and value to string translations, respectively, and `validate()` may be called to verify that a given value adheres to any restrictions on the values allowed for the type. The `errstr` element in the *`struct confd_type_ctx *ctx`* passed to these functions can be used to return an error message when the function fails - in this case `errstr` must be set to the address of a dynamically allocated string. The other elements in *`ctx`* are currently unused.

Including user-defined types in a YANG `union` may need some special consideration. Per the YANG specification, the string form of a value is matched against the union member types in the order they are specified until a match is found, and this procedure determines the type of the value. A corresponding procedure is used by ConfD when the value needs to be converted to a string, but this conversion does not include any evaluation of restrictions etc - the values are assumed to be correct for their type. Thus the `val_to_str()` function for the member types are tried in order until one succeeds, and the resulting string is used. This means that a) `val_to_str()` must verify that the value is of the correct type, i.e. that it has the expected confd_vtype, and b) if the value representation is the same for multiple member types, there is no guarantee that the same member type as for the string to value conversion is chosen.

The `opaque` element in the struct confd_type can be used for any auxiliary (static) data needed by the functions (on invocation they can reference it as self->opaque). The `parent` and `defval` elements are not used in this context, and should be NULL.

Note

The `str_to_val()` function *must* allocate space (using e.g. malloc(3)) for the actual data value for those confd_value_t types that are listed as having allocated data above, i.e. C_BUF, C_QNAME, C_LIST, C_OBJECTREF, C_OID, C_BINARY, and C_HEXSTR.

We make the implementation available to ConfD by creating one or more shared objects (.so files) containing the above callback functions. Each shared object may implement one or more types, and at startup the ConfD daemon will search the directories specified for `/confdConfig/loadPath` in `confd.conf` for files with a name that match the pattern "confd_type*.so" and load them.

Each shared object must also implement an "init" callback:

```
int confd_type_cb_init(struct confd_type_cbs **cbs);
```

When the object has been loaded, ConfD will call this function. It must return a pointer to an array of type callback structures via the *`cbs`* argument, and the number of elements in the array as return value. The struct confd_type_cbs is defined as:

```
struct confd_type_cbs {
    char *typepoint;
    struct confd_type *type;
};
```

These structures are then used by ConfD to locate the implementation of a given type, by searching for a `typepoint` string that matches the `tailf:typepoint` argument in the YANG data model.

Note

Since our callbacks are executed directly by the ConfD daemon, it is critically important that they do not have a negative impact on the daemon. No other processing can be done by ConfD while the callbacks are executed, and e.g. a NULL pointer dereference in one of the callbacks will cause ConfD to crash. Thus they should be simple, purely algorithmic functions, never referencing any external resources.

Note

When user-defined types are present, the ConfD daemon also needs to load the libconfd.so shared library, otherwise used only by applications. This means that either this library must be in one of the system directories that are searched by the OS runtime loader (typically /lib and /usr/lib), or its location must be given by setting the LD_LIBRARY_PATH environment variable before starting ConfD.

The above is enough for ConfD to use the types that we have defined, but the libconfd library can also do local string<->value translation if we have loaded the schema information, as described in the USING SCHEMA INFORMATION section below. For this to work for user-defined types, we must register the type definitions with the library, using one of these functions:

```
int confd_register_ns_type(u_int32_t nshash, const char *name, struct
confd_type *type);
```

Here we must pass the hash value for the namespace where the type is defined as *nshash*, and the name of the type from a `typedef` statement (i.e. *not* the typepoint name if they are different) as *name*. Thus we can not use this function to register a user-defined type that is specified "inline" in a `leaf` or `leaf-list` statement, since we don't have a name for the type.

```
int confd_register_node_type(struct confd_cs_node *node, struct
confd_type *type);
```

This function takes a pointer to a schema node (see the section USING SCHEMA INFORMATION) that uses the type instead of namespace and type name. It is necessary to use this for registration of user-defined types that are specified "inline", but it can also be used for user-defined types specified via `typedef`. In the latter case it will be equivalent to calling `confd_register_ns_type()` for the typedef, i.e. a single registration will apply to all nodes using the typedef.

The functions can only be called *after* `confd_load_schemas()` or `maapi_load_schemas()` (see below) has been called, and if `confd_load_schemas()`/`maapi_load_schemas()` is called again, the registration must be re-done. The `misc/user_type` example shows a way to use the exact same code for the shared object and for this registration.

Schema upgrades when the data is stored in CDB requires special consideration for user-defined types. Normally CDB can handle any type changes automatically, and this is true also when changing to/from/between user-defined types, provided that the following requirements are fulfilled:

1. A given typepoint name always refers to the exact same implementation - i.e. same value representation, same range restrictions, etc.

2. Shared objects providing implementations for all the typepoint ids used in the new *and* the old schema are made available to ConfD.

I.e. if we change the implementation of a type, we also change the typepoint name, and keep the old implementation around. If requirement 1 isn't fulfilled, we can end up with the case of e.g. a changed value representation between schema versions even though the types are indistinguishable for CDB. This can still be handled by using MAAPI to modify CDB during the upgrade as described in the User Guide, but if that is not done, CDB will just carry the old values over, which in effect results in a corrupt database.

USING SCHEMA INFORMATION

Schema information from the data model can be loaded from the ConfD daemon at runtime using the `maapi_load_schemas()` function, see the confd_lib_maapi(3) manual page. Information for all namespaces loaded into ConfD is then made available. In many cases it may be more convenient to use the

`confd_load_schemas()` utility function. For details about this function and those discussed below, see confd_lib_lib(3). After loading the data, we can call `confd_get_nslist()` to find which namespaces are known to the library as a result.

Note that all pointers returned (directly or indirectly) by the functions discussed here reference dynamically allocated memory maintained by the library - they will become invalid if `confd_load_schemas()` or `maapi_load_schemas()` is subsequently called again.

The confdc(1) compiler can also optionally generate a C header file that has #define symbols for the integer values corresponding to data model nodes and enumerations.

When the schema information has been made available to the library, we can format an arbitrary instance of a confd_value_t value using `confd_pp_value()` or `confd_ns_pp_value()`, or an arbitrary hkeypath using `confd_pp_kpath()` or `confd_xpath_pp_kpath()`. We can also get a pointer to the string representing a data model node using `confd_hash2str()`.

Furthermore a tree representation of the data model is available, which contains a struct confd_cs_node for every node in the data model. There is one tree for each namespace that has toplevel elements.

```
/* flag bits in confd_cs_node_info */
#define CS_NODE_IS_LIST          (1 << 0)
#define CS_NODE_IS_WRITE         (1 << 1)
#define CS_NODE_IS_CDB           (1 << 2)
#define CS_NODE_IS_ACTION        (1 << 3)
#define CS_NODE_IS_PARAM         (1 << 4)
#define CS_NODE_IS_RESULT        (1 << 5)
#define CS_NODE_IS_NOTIF         (1 << 6)
#define CS_NODE_IS_CASE          (1 << 7)
#define CS_NODE_IS_CONTAINER     (1 << 8)
#define CS_NODE_HAS_WHEN         (1 << 9)
#define CS_NODE_HAS_DISPLAY_WHEN (1 << 10)
#define CS_NODE_IS_DYN CS_NODE_IS_LIST /* backwards compat */

/* cmp values in confd_cs_node_info */
#define CS_NODE_CMP_NORMAL        0
#define CS_NODE_CMP_SNMP          1
#define CS_NODE_CMP_SNMP_IMPLIED  2
#define CS_NODE_CMP_USER          3
#define CS_NODE_CMP_UNSORTED      4

struct confd_cs_node_info {
    u_int32_t *keys;
    int minOccurs;
    int maxOccurs;   /* -1 if unbounded */
    enum confd_vtype shallow_type;
    struct confd_type *type;
    confd_value_t *defval;
    struct confd_cs_choice *choices;
    int flags;
    u_int8_t cmp;
};

struct confd_cs_node {
    u_int32_t tag;
    u_int32_t ns;
    struct confd_cs_node_info info;
    struct confd_cs_node *parent;
    struct confd_cs_node *children;
    struct confd_cs_node *next;
```

```
    void *opaque;   /* private user data */
};

struct confd_cs_choice {
    u_int32_t tag;
    u_int32_t ns;
    int minOccurs;
    struct confd_cs_case *default_case;
    struct confd_cs_node *parent;         /* NULL if parent is case */
    struct confd_cs_case *cases;
    struct confd_cs_choice *next;
    struct confd_cs_case *case_parent;    /* NULL if parent is node */
};

struct confd_cs_case {
    u_int32_t tag;
    u_int32_t ns;
    struct confd_cs_node *first;
    struct confd_cs_node *last;
    struct confd_cs_choice *parent;
    struct confd_cs_case *next;
    struct confd_cs_choice *choices;
};
```

Each confd_cs_node is linked to its related nodes: `parent` is a pointer to the parent node, `next` is a pointer to the next sibling node, and `children` is a pointer to the first child node - for each of these, a NULL pointer has the obvious meaning.

Each confd_cs_node also contains an information structure: For a list node, the `keys` field is a zero-terminated array of integers - these are the `tag` values for the children nodes that are key elements. This makes it possible to find the name of a key element in a keypath. If the confd_cs_node is not a list node, the `keys` field is NULL. The `shallow_type` field gives the "primitive" type for the element, i.e. the enum confd_vtype value that is used in the confd_value_t representation.

Typed leaf nodes also carry a complete type definition via the `type` pointer, which can be used with the `conf_str2val()` and `confd_val2str()` functions, as well as the leaf's default value (if any) via the `defval` pointer.

If the YANG `choice` statement is used in the data model, additional structures are created by the schema loading. For list and container nodes that have `choice` statements, the `choices` element in confd_cs_node_info is a pointer to a linked list of confd_cs_choice structures representing the choices. Each confd_cs_choice has a pointer to the parent node and a `cases` pointer to a linked list of confd_cs_case structures representing the cases for that choice. Finally, each confd_cs_case structure has pointers to the parent confd_cs_choice structure, and to the confd_cs_node structures representing the first and last element in the case. Those confd_cs_node structures, i.e. the "toplevel" elements of a case, have the CS_NODE_IS_CASE flag set. Note that it is possible for a case to be "empty", i.e. there are no elements in the case - then the `first` and `last` pointers in the confd_cs_case structure are NULL.

For a list node, the sort order is indicated by the `cmp` element in confd_cs_node_info. The value CS_NODE_CMP_NORMAL means an ordinary, system ordered, list. CS_NODE_CMP_SNMP is system ordered, but ordered according to SNMP lexicographical order, and CS_NODE_CMP_SNMP_IMPLIED is an SNMP lexicographical order where the last key has an IMPLIED keyword. CS_NODE_CMP_UNSORTED is system ordered, but is not sorted. The value CS_NODE_CMP_USER denotes an "ordered-by user" list.

Action and notification specifications are included in the tree in the same way as the config/data elements - they are indicated by the CS_NODE_IS_ACTION flag being set on the `action` node, and the

CS_NODE_IS_NOTIF flag being set on the `notification` node, respectively. Furthermore the nodes corresponding to the sub-statements of the action's `input` statement have the CS_NODE_IS_PARAM flag set, and those corresponding to the sub-statements of the action's `output` statement have the CS_NODE_IS_RESULT flag set. Note that the `input` and `output` statements do not have corresponding nodes in the tree.

The `confd_find_cs_root()` function returns the root of the tree for a given namespace, and the `confd_find_cs_node()`, `confd_find_cs_node_child()`, and `confd_cs_node_cd()` functions are useful for navigating the tree. Assume that we have the following data model:

```
container servers {
  list server {
    key name;
    max-elements 64;
    leaf name {
      type string;
    }
    leaf ip {
      type inet:ip-address;
    }
    leaf port {
      type inet:port-number;
    }
  }
}
```

Then, given the keypath `/servers/server{www}` in confd_hkeypath_t form, a call to `confd_find_cs_node()` would return a struct confd_cs_node, i.e. a pointer into the tree, as in:

```
struct confd_cs_node *csp;
char *name;
csp = confd_find_cs_node(mykeypath, mykeypath->len);
name = confd_hash2str(csp->info.keys[0])
```

and the C variable `name` will have the value `"name"`. These functions make it possible to format keypaths in various ways.

If we have a keypath which identifies a node below the one we are interested in, such as `/servers/server{www}/ip`, we can use the *`len`* parameter as in `confd_find_cs_node(kp, 3)` where `3` is the length of the keypath we wish to consider.

The equivalent of the above `confd_find_cs_node()` example, but using a string keypath, could be written as:

```
csp = confd_cs_node_cd(confd_find_cs_root(mynamespace),
                       "/servers/server{www}");
```

The `type` field in the struct confd_cs_node_info can be used for data model aware string <-> value translations. E.g. assuming that we have a confd_hkeypath_t *kp representing the element `/servers/server{www}/ip`, we can do the following:

```
confd_value_t v;
csp = confd_find_cs_node(kp, kp->len);
confd_str2val(csp->info.type, "10.0.0.1", &v);
```

The confd_value_t v will then be filled in with the corresponding C_IPV4 value. This technique is generally necessary for translating C_ENUM_VALUE values to the corresponding strings (or vice versa), since

there isn't a type-independent mapping. But `confd_val2str()` (or `confd_str2val()`) can always do the translation, since it is given the full type information. E.g. this will store the string "nonVolatile" in `buf`:

```
confd_value_t v;
char buf[64];

CONFD_SET_ENUM_VALUE(&v, 3);
root = confd_find_cs_root(SNMP_COMMUNITY_MIB__ns);
csp = confd_cs_node_cd(root, "/SNMP-COMMUNITY-MIB/snmpCommunityTable/"
                       "snmpCommunityEntry/snmpCommunityStorageType");
confd_val2str(csp->info.type, &v, buf, sizeof(buf));
```

The type information can also be found by using the `confd_find_ns_type()` function to look up the type name as a string in the namespace where it is defined - i.e. we could alternatively have achieved the same result with:

```
CONFD_SET_ENUM_VALUE(&v, 3);
type = confd_find_ns_type(SNMPv2_TC__ns, "StorageType");
confd_val2str(type, &v, buf, sizeof(buf));
```

If we give `0` for the *`nshash`* argument to `confd_find_ns_type()`, the type name will be looked up among the ConfD built-in types (i.e. the YANG built-in types, the types defined in the YANG "tailf-common" module, and the types defined in the pre-defined "confd" and/or "xs" namespaces) - e.g. the type information for `/servers/server{www}/name` could be found with `confd_find_ns_type(0, "string")`.

XML STRUCTURES

Two different methods are used to represent a subtree of data nodes. "Value Array" describes a format that is simpler but has some limitations, while "Tagged Value Array" describes a format that is more complex but can represent an arbitrary subtree.

Value Array

The simpler format is an array of confd_value_t elements corresponding to the complete contents of a list entry or container. The content of sub-list entries cannot be represented. The array is populated through a "depth first" traversal of the data tree as follows:

1. Optional leafs or `presence` containers that do not exist use a single array element, with type C_NOEXISTS (value ignored).

2. List nodes use a single array element, with type C_NOEXISTS (value ignored), regardless of the actual number of entries or their contents.

3. Leafs with a type other than empty use an array element with their type and value as usual.

4. Leafs of type empty use an array element with type C_XMLTAG, and `tag` and `ns` set according to the leaf name.

5. Containers use one array element with type C_XMLTAG, and `tag` and `ns` set according to the element name, followed by array elements for the sub-nodes according to this list.

Note that the list or container node corresponding to the complete array is not included in the array, and that there is no array element for the "end" of a container.

As an example, the array corresponding to the `/servers/server{www}` list entry above could be populated as:

```
confd_value_t v[3];
struct in_addr ip;

CONFD_SET_STR(&v[0], "www");
ip.s_addr = inet_addr("192.168.1.2");
CONFD_SET_IPV4(&v[1], ip);
CONFD_SET_UINT16(&v[2], 80);
```

Tagged Value Array

This format uses an array of confd_tag_value_t elements. This is a structure defined as:

```
typedef struct confd_tag_value {
    struct xml_tag tag;
    confd_value_t v;
} confd_tag_value_t;
```

I.e. each value element is associated with the struct xml_tag that identifies the node in the data model. The `ns` element of the struct xml_tag can normally be set to 0, with the meaning "current namespace". The array is populated, normally through a "depth first" traversal of the data tree, as follows:

1. Optional leafs or `presence` containers that do not exist are omitted entirely from the array.

2. List and container nodes use one array element where the value has type C_XMLBEGIN, and `tag` and `ns` set according to the node name, followed by array elements for the sub-nodes according to this list, followed by one array element where the value has type C_XMLEND, and `tag` and `ns` set according to the node name.

3. Leafs with a type other than empty use an array element with their type and value as usual.

4. Leafs of type empty use an array element where the value has type C_XMLTAG, and `tag` and `ns` set according to the leaf name.

Note that the list or container node corresponding to the complete array is not included in the array. In some usages, non-optional nodes may also be omitted from the array - refer to the relevant API documentation to see whether this is allowed and the semantics of doing so.

A set of CONFD_SET_TAG_XXX() macros corresponding to the CONFD_SET_XXX() macros described above are provided - these set the `ns` element to 0 and the `tag` element to their second argument. The array corresponding to the `/servers/server{www}` list entry above could be populated as:

```
confd_tag_value_t tv[3];
struct in_addr ip;

CONFD_SET_TAG_STR(&tv[0], servers_name, "www");
ip.s_addr = inet_addr("192.168.1.2");
CONFD_SET_TAG_IPV4(&tv[1], servers_ip, ip);
CONFD_SET_TAG_UINT16(&tv[2], servers_port, 80);
```

There are also macros to access the components of the confd_tag_value_t elements:

```
confd_tag_value_t tv;
u_int16_t port;

if (CONFD_GET_TAG_TAG(&tv) == servers_port)
```

```
    port = CONFD_GET_UINT16(CONFD_GET_TAG_VALUE(&tv));
```

DATA MODEL TYPES

This section describes the types that can be used in YANG data modeling, and their C representation. Also listed is the corresponding SMIv2 type, which is used when a data model is translated into a MIB. In several cases, the data model type cannot easily be translated into a native SMIv2 type. In those cases, the type OCTET STRING is used in the translation. The SNMP agent in ConfD will in those cases send the string representation of the value over SNMP. For example, the xs:float value `3.14` is sent as the string "3.14".

These subsections describe the following sets of types, which can be used with YANG data modeling:

- YANG built-in types
- The ietf-yang-types YANG module
- The ietf-inet-types YANG module
- The tailf-common YANG module
- The tailf-xsd-types YANG module

YANG built-in types

These types are built-in to the YANG language, and also built-in to ConfD.

int8 — A signed 8-bit integer.

- `value.type` = C_INT8
- union element = `i8`
- C type = int8_t
- SMIv2 type = Integer32 (-128 .. 127)

int16 — A signed 16-bit integer.

- `value.type` = C_INT16
- union element = `i16`
- C type = int16_t
- SMIv2 type = Integer32 (-32768 .. 32767)

int32 — A signed 32-bit integer.

- `value.type` = C_INT32
- union element = `i32`
- C type = int32_t
- SMIv2 type = Integer32

int64 — A signed 64-bit integer.

- `value.type` = C_INT64
- union element = `i64`
- C type = int64_t
- SMIv2 type = OCTET STRING

uint8 — An unsigned 8-bit integer.

- `value.type` = C_UINT8
- union element = `u8`
- C type = u_int8_t
- SMIv2 type = Unsigned32 (0 .. 255)

uint16 — An unsigned 16-bit integer.

- `value.type` = C_UINT16
- union element = `u16`
- C type = u_int16_t
- SMIv2 type = Unsigned32 (0 .. 65535)

uint32 — An unsigned 32-bit integer.

- `value.type` = C_UINT32
- union element = `u32`
- C type = u_int32_t
- SMIv2 type = Unsigned32

uint64 — An unsigned 64-bit integer.

- `value.type` = C_UINT64
- union element = `u64`
- C type = u_int64_t
- SMIv2 type = OCTET STRING

decimal64 — A decimal number with 64 bits of precision. The C representation uses a struct with a 64-bit signed integer for the scaled value, and an unsigned 8-bit integer in the range 1..18 for the number of fraction digits specified by the `fraction-digits` sub-statement.

- `value.type` = C_DECIMAL64
- union element = `d64`
- C type = struct confd_decimal64

- SMIv2 type = OCTET STRING

string

The string type is represented as a struct confd_buf_t when *received* from ConfD in the C code. I.e. it is NUL-terminated and also has a size given.

However, when the C code wants to produce a value of the string type it is possible to use a confd_value_t with the value type C_BUF or C_STR (which requires a NUL-terminated string)

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

boolean

The boolean values "true" and "false".

- `value.type` = C_BOOL
- union element = `boolean`
- C type = int
- SMIv2 type = TruthValue

enumeration

Enumerated strings with associated numeric values. The C representation uses the numeric values.

- `value.type` = C_ENUM_VALUE
- union element = `enumvalue`
- C type = int32_t
- SMIv2 type = INTEGER

bits

A set of bits or flags. Depending on the highest argument given to a `position` sub-statement, the C representation uses either C_BIT32 or C_BIT64.

- `value.type` = C_BIT32 or C_BIT64
- union element = `b32` or `b64`
- C type = u_int32_t or u_int64_t
- SMIv2 type = Unsigned32 or OCTET STRING

binary

Any binary data.

- `value.type` = C_BINARY
- union element = `buf`
- C type = confd_buf_t

- SMIv2 type = OCTET STRING

identityref — A reference to an abstract identity.

- `value.type` = C_IDENTITYREF
- union element = `idref`
- C type = struct confd_identityref
- SMIv2 type = OCTET STRING

union — The union type has no special confd_value_t representation - elements are represented as one of the member types according to the current value instantiation. This means that for unions that comprise different "primitive" types, applications must check the `type` element to determine the type, and the type safe alternatives to the `cdb_get()` and `maapi_get_elem()` functions can not be used.

The SMIv2 type is an OCTET STRING.

instance-identifier — The instance-identifier built-in type is used to uniquely identify a particular instance node in the data tree. The syntax for an instance-identifier is a subset of the XPath abbreviated syntax.

- `value.type` = C_OBJECTREF
- union element = `hkp`
- C type = confd_hkeypath_t
- SMIv2 type = OCTET STRING

The `leaf-list` statement

The values of a YANG `leaf-list` node is represented as an element with a list of values of the type given by the `type` sub-statement.

- `value.type` = C_LIST
- union element = `list`
- C type = struct confd_list
- SMIv2 type = OCTET STRING

The ietf-yang-types YANG module

This module contains a collection of generally useful derived YANG data types. They are defined in the `urn:ietf:params:xml:ns:yang:ietf-yang-types` namespace.

yang:counter32, yang:zero-based-counter32 — 32-bit counters, corresponding to the Counter32 type and the ZeroBasedCounter32 textual convention of the SMIv2.

- `value.type` = C_UINT32
- union element = `u32`

	• C type = u_int32_t • SMIv2 type = Counter32
yang:counter64, yang:zero-based-counter64	64-bit counters, corresponding to the Counter64 type and the ZeroBasedCounter64 textual convention of the SMIv2. • `value.type` = C_UINT64 • union element = `u64` • C type = u_int64_t • SMIv2 type = Counter64
yang:gauge32	32-bit gauge value, corresponding to the Gauge32 type of the SMIv2. • `value.type` = C_UINT32 • union element = `u32` • C type = u_int32_t • SMIv2 type = Counter32
yang:gauge64	64-bit gauge value, corresponding to the CounterBasedGauge64 SMIv2 textual convention. • `value.type` = C_UINT64 • union element = `u64` • C type = u_int64_t • SMIv2 type = Counter64
yang:object-identifier, yang:object-identifier-128	An SNMP OBJECT IDENTIFIER (OID). This is a sequence of integers which identifies an object instance for example "1.3.6.1.4.1.24961.1". **Note** The `tailf:value-length` restriction is measured in integer elements for object-identifier and object-identifier-128. • `value.type` = C_OID • union element = `oidp` • C type = confd_snmp_oid • SMIv2 type = OBJECT IDENTIFIER
yang:yang-identifier	A YANG identifier string as defined by the 'identifier' rule in Section 12 of RFC 6020.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

yang:date-and-time

The date-and-time type is a profile of the ISO 8601 standard for representation of dates and times using the Gregorian calendar.

- `value.type` = C_DATETIME
- union element = `datetime`
- C type = struct confd_datetime
- SMIv2 type = DateAndTime

yang:timeticks, yang:timestamp

Time ticks and time stamps, measured in hundredths of seconds. Corresponding to the TimeTicks type and the TimeStamp textual convention of the SMIv2.

- `value.type` = C_UINT32
- union element = `u32`
- C type = u_int32_t
- SMIv2 type = Counter32

yang:phys-address

Represents media- or physical-level addresses represented as a sequence octets, each octet represented by two hexadecimal digits. Octets are separated by colons.

Note

The `tailf:value-length` restriction is measured in number of octets for phys-address.

- `value.type` = C_BINARY
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

yang:mac-address

The mac-address type represents an IEEE 802 MAC address.

The length of the ConfD C_BINARY representation is always 6.

- `value.type` = C_BINARY
- union element = `buf`
- C type = confd_buf_t

- SMIv2 type = OCTET STRING

yang:xpath1.0 — This type represents an XPATH 1.0 expression.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

yang:hex-string — A hexadecimal string with octets represented as hex digits separated by colons.

> **Note**
>
> The `tailf:value-length` restriction is measured in number of octets for hex-string.

- `value.type` = C_HEXSTR
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

yang:uuid — A Universally Unique Identifier in the string representation defined in RFC 4122.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

yang:dotted-quad — An unsigned 32-bit number expressed in the dotted-quad notation.

- `value.type` = C_DQUAD
- union element = `dquad`
- C type = struct confd_dotted_quad
- SMIv2 type = OCTET STRING

The ietf-inet-types YANG module

This module contains a collection of generally useful derived YANG data types for Internet addresses and related things. They are defined in the `urn:ietf:params:xml:ns:yang:inet-types` name-space.

inet:ip-version — This value represents the version of the IP protocol.

- `value.type` = C_ENUM_VALUE
- union element = `enumvalue`
- C type = int32_t
- SMIv2 type = INTEGER

inet:dscp — The dscp type represents a Differentiated Services Code-Point.

- `value.type` = C_UINT8
- union element = `u8`
- C type = u_int8_t
- SMIv2 type = Unsigned32 (0 .. 255)

inet:ipv6-flow-label — The flow-label type represents flow identifier or Flow Label in an IPv6 packet header.

- `value.type` = C_UINT32
- union element = `u32`
- C type = u_int32_t
- SMIv2 type = Unsigned32

inet:port-number — The port-number type represents a 16-bit port number of an Internet transport layer protocol such as UDP, TCP, DCCP or SCTP.

The value space and representation is identical to the built-in uint16 type.

inet:as-number — The as-number type represents autonomous system numbers which identify an Autonomous System (AS).

The value space and representation is identical to the built-in uint32 type.

inet:ip-address — The ip-address type represents an IP address and is IP version neutral. The format of the textual representations implies the IP version.

This is a union of the inet:ipv4-address and inet:ipv6-address types defined below. The representation is thus identical to the representation for one of these types.

The SMIv2 type is an OCTET STRING (SIZE (4|16)).

inet:ipv4-address — The ipv4-address type represents an IPv4 address in dotted-quad notation.

The use of a zone index is not supported by ConfD.

- `value.type` = C_IPV4
- union element = `ip`
- C type = struct in_addr

- SMIv2 type = IpAddress

inet:ipv6-address

The ipv6-address type represents an IPv6 address in full, mixed, shortened and shortened mixed notation.

The use of a zone index is not supported by ConfD.

- `value.type` = C_IPV6
- union element = `ip6`
- C type = struct in6_addr
- SMIv2 type = IPV6-MIB:Ipv6Address

inet:ip-prefix

The ip-prefix type represents an IP prefix and is IP version neutral. The format of the textual representations implies the IP version.

This is a union of the inet:ipv4-prefix and inet:ipv6-prefix types defined below. The representation is thus identical to the representation for one of these types.

The SMIv2 type is an OCTET STRING (SIZE (5|17)).

inet:ipv4-prefix

The ipv4-prefix type represents an IPv4 address prefix. The prefix length is given by the number following the slash character and must be less than or equal to 32.

A prefix length value of n corresponds to an IP address mask which has n contiguous 1-bits from the most significant bit (MSB) and all other bits set to 0.

The IPv4 address represented in dotted quad notation must have all bits that do not belong to the prefix set to zero.

An example: 10.0.0.0/8

- `value.type` = C_IPV4PREFIX
- union element = `ipv4prefix`
- C type = struct confd_ipv4_prefix
- SMIv2 type = OCTET STRING (SIZE (5))

inet:ipv6-prefix

The ipv6-prefix type represents an IPv6 address prefix. The prefix length is given by the number following the slash character and must be less than or equal 128.

A prefix length value of n corresponds to an IP address mask which has n contiguous 1-bits from the most significant bit (MSB) and all other bits set to 0.

The IPv6 address must have all bits that do not belong to the prefix set to zero.

An example: 2001:DB8::1428:57AB/125

- `value.type` = C_IPV6PREFIX
- union element = `ipv6prefix`
- C type = struct confd_ipv6_prefix
- SMIv2 type = OCTET STRING (SIZE (17))

inet:domain-name — The domain-name type represents a DNS domain name. The name SHOULD be fully qualified whenever possible.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

inet:host — The host type represents either an IP address or a DNS domain name.

This is a union of the inet:ip-address and inet:domain-name types defined above. The representation is thus identical to the representation for one of these types.

The SMIv2 type is an OCTET STRING, which contains the textual representation of the domain name or address.

inet:uri — The uri type represents a Uniform Resource Identifier (URI) as defined by STD 66.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

The iana-crypt-hash YANG module

This module defines a type for storing passwords using a hash function, and features to indicate which hash functions are supported by an implementation. The type is defined in the `urn:ietf:params:xml:ns:yang:iana-crypt-hash` namespace.

ianach:crypt-hash — The crypt-hash type is used to store passwords using a hash function. The algorithms for applying the hash function and encoding the result are implemented in various UNIX systems as the function crypt(3). A value of this type matches one of the forms:

```
$0$<clear text password>
$<id>$<salt>$<password hash>
$<id>$<parameter>$<salt>$<password hash>
```

The "0" prefix signals that the value is clear text. When such a value is received by the server, a hash value is calculated, and the string "$<id>$<salt>

$" or $<id>$<parameter>$<salt>$ is prepended to the result. This value is stored in the configuration data store.

If a value starting with "$<id>$", where <id> is not "0", is received, the server knows that the value already represents a hashed value, and stores it as is in the data store.

In the Tail-f implementation, this type is logically a union of the types tailf:md5-digest-string, tailf:sha-256-digest-string, and tailf:sha-512-digest-string - see the section The tailf-common YANG module below. All the hashed values of these types are accepted, and the choice of algorithm to use for hashing clear text is specified via the `/confdConfig/cryptHash/algorithm` parameter in `confd.conf` (see confd.conf(5)). If the algorithm is set to "sha-256" or "sha-512", it can be tuned via the `/confdConfig/cryptHash/rounds` parameter in `confd.conf`.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

The tailf-common YANG module

This module defines Tail-f common YANG types, that are built-in to ConfD.

tailf:size

A value that represents a number of bytes. An example could be S1G8M7K956B; meaning 1GB+8MB+7KB+956B = 1082138556 bytes. The value must start with an S. Any byte magnifier can be left out, i.e. S1K1B equals 1025 bytes. The order is significant though, i.e. S1B56G is not a valid byte size.

The value space and representation is identical to the built-in uint64 type.

tailf:octet-list

A list of dot-separated octets for example "192.168.255.1.0".

Note

The `tailf:value-length` restriction is measured in number of octets for octet-list.

- `value.type` = C_BINARY
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:hex-list

A list of colon-separated hexa-decimal octets for example "4F:4C:41:71".

Note

The `tailf:value-length` restriction is measured in octets of binary data for hex-list.

- `value.type` = C_BINARY
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:md5-digest-string

The md5-digest-string type automatically computes a MD5 digest for a value adhering to this type.

This is best explained using an example. Suppose we have a leaf:

```
leaf key {
  type tailf:md5-digest-string;
}
```

A valid configuration is:

```
<key>$0$In god we trust.</key>
```

The "0" prefix indicates that this is plain text and that this value should be represented as a MD5 digest from now. ConfD computes a MD5 digest for the value and prepends "1<salt>$", where <salt> is a random eight character salt used to generate the digest. When this value later on is fetched from ConfD the following is returned:

```
<key>$1$fB$ndk2z/PIS0S1SvzWLqTJb.</key>
```

A value adhering to md5-digest-string must have "0" or a "1<salt>$" prefix.

The digest algorithm is the same as the md5 crypt function used for encrypting passwords for various UNIX systems, e.g. http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libcrypt/crypt.c?rev=1.5&content-type=text/plain

Note

The `pattern` restriction can not be used with this type.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t

- SMIv2 type = OCTET STRING

tailf:sha-256-digest-string

The sha-256-digest-string type automatically computes a SHA-256 digest for a value adhering to this type. A value of this type matches one of the forms:

```
$0$<clear text password>
$5$<salt>$<password hash>
$5$rounds=<number>$<salt>$<password hash>
```

The "0" prefix signals that this is plain text. When a plain text value is received by the server, a SHA-256 digest is calculated, and the string "5<salt>$" is prepended to the result, where <salt> is a random 16 character salt used to generate the digest. This value is stored in the configuration data store. The algorithm can be tuned via the `/confdConfig/cryptHash/rounds` parameter in `confd.conf` (see confd.conf(5)), which if set to a number other than the default will cause "5rounds=<number>$<salt>$" to be prepended instead of only "5<salt>$".

If a value starting with "5" is received, the server knows that the value already represents a SHA-256 digest, and stores it as is in the data store.

The digest algorithm used is the same as the SHA-256 crypt function used for encrypting passwords for various UNIX systems, see e.g. http://www.akkadia.org/drepper/SHA-crypt.txt

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:sha-512-digest-string

The sha-512-digest-string type automatically computes a SHA-512 digest for a value adhering to this type. A value of this type matches one of the forms:

```
$0$<clear text password>
$6$<salt>$<password hash>
$6$rounds=<number>$<salt>$<password hash>
```

The "0" prefix signals that this is plain text. When a plain text value is received by the server, a SHA-512 digest is calculated, and the string "6<salt>$" is prepended to the result, where <salt> is a random 16 character salt used to generate the digest. This value is stored in the configuration data store. The algorithm can be tuned via the `/confdConfig/cryptHash/rounds` parameter in `confd.conf` (see confd.conf(5)), which if set to a number other than the default will cause "6rounds=<number>$<salt>$" to be prepended instead of only "6<salt>$".

If a value starting with "6" is received, the server knows that the value already represents a SHA-512 digest, and stores it as is in the data store.

The digest algorithm used is the same as the SHA-512 crypt function used for encrypting passwords for various UNIX systems, see e.g. http://www.akkadia.org/drepper/SHA-crypt.txt

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:des3-cbc-encrypted-string

The des3-cbc-encrypted-string type automatically encrypts a value adhering to this type using DES in CBC mode followed by a base64 conversion. If the value isn't encrypted already, that is.

This is best explained using an example. Suppose we have a leaf:

```
leaf enc {
  type tailf:des3-cbc-encrypted-string;
}
```

A valid configuration is:

```
<enc>$0$In god we trust.</enc>
```

The "0" prefix indicates that this is plain text and that this value should be DES3/Base64 encrypted from now. ConfD encrypts the value and prepends "3". When this value later on is fetched from ConfD the following is returned:

```
<enc>$3$lyPjszaQq4EVqK7OPOxybQ==</enc>
```

A value adhering to des3-cbc-encrypted-string must have "0" or a "3" prefix.

ConfD uses configurable set of encryption keys to encrypt the string. If we want our own custom pass phrase this can be configured as described by the encryptedStrings configurable in the confd.conf(5) manual page.

Note

The `pattern` restriction can not be used with this type.

- `value.type` = C_BUF
- union element = `buf`

- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:aes-cfb-128-encrypted-string

The aes-cfb-128-encrypted-string works exactly like des3-cbc-encrypted-string but AES/128bits in CFB mode is used to encrypt the string. The prefix for encrypted values is "4".

Note

The `pattern` restriction can not be used with this type.

- `value.type` = C_BUF
- union element = `buf`
- C type = confd_buf_t
- SMIv2 type = OCTET STRING

tailf:ip-address-and-prefix-length

The ip-address-and-prefix-length type represents a combination of an IP address and a prefix length and is IP version neutral. The format of the textual representations implies the IP version.

This is a union of the tailf:ipv4-address-and-prefix-length and tailf:ipv6-address-and-prefix-length types defined below. The representation is thus identical to the representation for one of these types.

The SMIv2 type is an OCTET STRING (SIZE (5|17)).

tailf:ipv4-address-and-prefix-length

The ipv4-address-and-prefix-length type represents a combination of an IPv4 address and a prefix length. The prefix length is given by the number following the slash character and must be less than or equal to 32.

An example: 172.16.1.2/16

- `value.type` = C_IPV4_AND_PLEN
- union element = `ipv4prefix`
- C type = struct confd_ipv4_prefix
- SMIv2 type = OCTET STRING (SIZE (5))

tailf:ipv6-address-and-prefix-length

The ipv6-address-and-prefix-length type represents a combination of an IPv6 address and a prefix length. The prefix length is given by the number following the slash character and must be less than or equal to 128.

An example: 2001:DB8::1428:57AB/64

- `value.type` = C_IPV6_AND_PLEN
- union element = `ipv6prefix`

- C type = struct confd_ipv6_prefix
- SMIv2 type = OCTET STRING (SIZE (17))

The tailf-xsd-types YANG module

"This module contains useful XML Schema Datatypes that are not covered by YANG types directly.

xs:duration	• `value.type` = C_DURATION • union element = `duration` • C type = struct confd_duration • SMIv2 type = OCTET STRING
xs:date	• `value.type` = C_DATE • union element = `date` • C type = struct confd_date • SMIv2 type = OCTET STRING
xs:time	• `value.type` = C_TIME • union element = `time` • C type = struct confd_time • SMIv2 type = OCTET STRING
xs:token	• `value.type` = C_BUF • union element = `buf` • C type = confd_buf_t • SMIv2 type = OCTET STRING
xs:hexBinary	• `value.type` = C_BINARY • union element = `buf` • C type = confd_buf_t • SMIv2 type = OCTET STRING
xs:QName	• `value.type` = C_QNAME • union element =`qname` • C type = struct confd_qname • SMIv2 type = <not applicable>
xs:decimal, xs:float, xs:double	• `value.type` = C_DOUBLE

- union element = d
- C type = double
- SMIv2 type = OCTET STRING

SEE ALSO

The ConfD User Guide

confd_lib(3) - confd C library.

confd.conf(5) - confd daemon configuration file format

ConfD man-pages, Volume 5

Table of Contents

Name

clispec — CLI specification file format

DESCRIPTION

This manual page describes the syntax and semantics of a ConfD CLI specification file (from now on called "clispec"). A clispec is an XML configuration file describing commands to be added to the automatically rendered Juniper and Cisco style ConfD CLI. It also makes it possible to modify the behavior of standard/built-in commands, using move/delete operations and customizable confirmation prompts. In Cisco style custom mode-specific commands can be added by specifying a mount point relating to the specified mode.

Tip

In the ConfD distribution there is an Emacs mode suitable for clispec editing.

A clispec file (with a .cli suffix) is to be compiled using the **confdc** compiler into an internal representation (with a .ccl suffix), ready to be loaded by the ConfD daemon on startup. Like this:

```
$ confdc -c commands.cli
$ ls commands.ccl
commands.ccl
```

The .ccl file should be put in the ConfD daemon loadPath as described in confd.conf(5) When the ConfD daemon is started the clispec is loaded accordingly.

The ConfD daemon loads all .ccl files it finds on startup. Ie, you can have one or more clispec files for Cisco XR (C) style CLI emulation, one or more for Cisco IOS (I), and one or more for Juniper (J) style emulation. If you drop several .ccl files in the loadPath all will be loaded. The standard commands are defined in confd.cli (available in the ConfD distribution). The intention is that we use confd.cli as a starting point, i.e. first we delete, reorder and replace built-in commands (if needed) and we then proceed to add our own custom commands.

EXAMPLE

The confd-light.cli example is a light version of the standard confd.cli. It adds one operational mode command and one configure mode command, implemented by two OS executables, it also removes the 'save' command from the pipe commands.

Example 147. confd-light.cli

```
<clispec xmlns="http://tail-f.com/ns/clispec/1.0" style="j">
  <operationalMode>
    <modifications>
      <delete src="file"/>
      <confirmText src="quit">
        Are you really sure you want to quit?
      </confirmText>
      <help src="configure private">Edit a private copy of the configuration</help>
      <info src="configure private">Edit a private copy of the configuration</info>
    </modifications>

    <cmd name="copy" mount="file">
```

```
      <info>Copy a file</info>
      <help>Copy a file in the file system.</help>
      <callback>
        <exec>
          <osCommand>cp</osCommand>
          <options>
            <uid>confd</uid>
          </options>
        </exec>
      </callback>
      <params>
        <param>
          <type><file/></type>
          <info>&lt;source file&gt;</info>
        </param>
        <param>
          <type><file/></type>
          <info>&lt;destination&gt;</info>
        </param>
      </params>
    </cmd>
  </operationalMode>

  <configureMode>
    <cmd name="adduser" mount="wizard">
      <info>Create a user</info>
      <help>Create a user and assign him/her to a group.</help>
      <callback>
        <exec>
          <osCommand>adduser.sh</osCommand>
        </exec>
      </callback>
    </cmd>
  </configureMode>

  <pipeCmds>
    <modifications>
      <delete src="save"/>
    </modifications>
  </pipeCmds>
</clispec>
```

confd-light.cli achieves the following:

- Adds a confirmation prompt to the standard operation "delete" command.
- Deletes the standard "file" command.
- Adds the operational mode command "copy" and mounts it under the standard "file" command.
- The "copy" command is implemented using the OS executable "/usr/bin/cp".
- The executable is called with parameters as defined by the "params" element.
- The executable runs as the same user id as ConfD as defined by the "uid" element.
- Adds the configure command "adduser" and mounts it under the standard "wizard" command.

Below we present the gory details when it comes to constructs in a clispec.

ELEMENTS AND ATTRIBUTES

This section lists all clispec elements and their attributes including their type (within parentheses) and default values (within square brackets). Elements are written using a path notation to make it easier to see how they relate to each other.

Note: $MODE is either "operationalMode", "configureMode" or "pipeCmds".

`/clispec`

This is the top level element which contains (in order) zero or more "operationalMode" elements, zero or more "configureMode" element, and zero or more "pipeCmds" elements.

It has a style attribute which can have the value "j", "i" or "c". If no style attribute is specified it defaults to "j".

`/clispec/$MODE`

The $MODE ("operationalMode", "configureMode", or "pipeCmds") element contains (in order) zero or one "modifications" elements, zero or more "start" elements, zero or more "show" elements, and zero or more "cmd" elements.

The "show" elements are only used in the C-style CLI.

It has a name attribute which is used to create a named custom mode. A custom command can be defined for entering custom modes. See the cmd/callback/mode elements below.

`/clispec/$MODE/modifications`

The "modifications" element describes which operations to apply to the built-in commands. It contains (in any order) zero or more "delete", "move", "paginate", "info", "paraminfo", "help", "paramhelp", "confirmText", "defaultConfirmOption", "dropElem", "compactElem", "compactStatsElem", "columnStats", "multiValue", "columnWidth", "columnAlign", "defaultColumnAlign", "noKeyCompletion", "noMatchCompletion", "modeName", "suppressMode", "suppressTable", "enforceTable", "showTemplate", "showTemplateLegend", "showTemplateEnter", "showTemplateFooter", "runTemplate", "runTemplateLegend", "runTemplateEnter", "runTemplateFooter", "addMode", "autocommitDelay", "keymap", "pipeFlags", "addPipeFlags", "negPipeFlags", "legend", "footer", "suppressKeyAbbrev", "allowKeyAbbrev", "hasRange", "suppressRange", "allowWildcard", "suppressWildcard", "suppressValidationWarningPrompt", "displayEmptyConfig", "displayWhen", "customRange", "completion", "keepKeyOrder" and "simpleType" elements.

`/clispec/$MODE/modifications/paginate`

The "paginate" element can be used to change the default paginate behavior for a built-in command.

Attributes:

path (cmdpathType)	The "path" attribute is mandatory. It specifies which command to change. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.
value (true|false)	The "value" attribute is mandatory. It specifies whether the paginate attribute should be enabled or disabled by default.

`/clispec/$MODE/modifications/displayWhen`

The "displayWhen" element can be used to add a displayWhen xpath condition to a command.

Attributes:

path (cmdpathType) — The "path" attribute is mandatory. It specifies which command to change. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

exrp (xpath expression) — The "expr" attribute is mandatory. It specifies an xpath expression. If the expression evaluates to true then the command is available, otherwise not.

ctx (path) — The "ctx" attribute is optional. If not specified the current editpath/mode-path is used as context node for the xpath evaluation. Note that the xpath expression will automatically evaluate to false if a display when expression is used for a top-level command and no ctx is specified. The path may contain variables defined in the dict.

`/clispec/$MODE/modifications/move`

The "move" element can be used to move (rename) a built-in command.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to move. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

dest (cmdpathType) — The "dest" attribute is mandatory. It specifies where to move the command specified by the "src" attribute. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

inclSubCmds (xs:boolean) — The "inclSubCmds" attribute is optional. If specified and set to true then all commands to which the 'src' command is a prefix command will be included in the move operation.

An example:

```
<configureMode>
  <modifications>
    <move src="load" dest="xload" inclSubCmds="t
  </modifications>
</configureMode>
```

would in the C-style CLI move 'load', 'load merge', 'load override' and 'load replace' to

'xload', 'xload merge', 'xload override' and 'xload replace', respectively.

`/clispec/$MODE/modifications/copy`

The "copy" element can be used to copy a built-in command.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to copy. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

dest (cmdpathType) — The "dest" attribute is mandatory. It specifies where to copy the command specified by the "src" attribute. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

inclSubCmds (xs:boolean) — The "inclSubCmds" attribute is optional. If specified and set to true then all commands to which the 'src' command is a prefix command will be included in the copy operation.

An example:

```
<configureMode>
  <modifications>
    <copy src="load" dest="xload" inclSubCmds="t
  </modifications>
</configureMode>
```

would in the C-style CLI copy 'load', 'load merge', 'load override' and 'load replace' to 'xload', 'xload merge', 'xload override' and 'xload replace', respectively.

`/clispec/$MODE/modifications/delete`

The "delete" element makes it possible to delete a built-in command. Note that commands that are auto-rendered from the data model cannot be removed using this modification. To remove an auto-rendered command use the 'tailf:hidden' element in the data model.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to delete. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/pipeFlags`

The "pipeFlags" element makes it possible to modify the pipe flags of the builtin commands. The argument is a space separated list of pipe flags. It will replace the builtin list.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to modify. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/addPipeFlags`

The "addPipeFlags" element makes it possible to add pipe flags to the existing list of pipe flags for a builtin command. The argument is a space separated list of pipe flags.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to modify. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/negPipeFlags`

The "negPipeFlags" element makes it possible to modify the neg pipe flags of the builtin commands. The argument is a space separated list of neg pipe flags. It will replace the builtin list.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to modify. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/dropElem`

The "dropElem" element makes it possible to drop node in the data model from paths in the Cisco style CLIs. If you drop a child node to a list node we recommend that you also use suppressMode on that list node, otherwise the CLI will be very confusing. For example, for the alias command in the CLI. If we only dropped the expansion node but did not suppress the automatic mode creation for the alias node, when you typed the alias command you would end up in the alias submode, but since you have dropped the expansion node you end up specifying the expansion directly without typing any command. Quite confusing.

Note that dropped nodes to not appear in match completions.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to drop. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-drop-node-name YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/compactElem`

The "compactElem" element tells the C- and I-style CLIs 'show running-configuration' command to use the compact representation for this path. The compact representation means that all leaf elements are shown on a single line.

Attributes:

src (pathType) — The "src" attribute is mandatory. It specifies which path to make compact. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-compact-syntax YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/compactStatsElem`

The "compactStatsElem" element tells the show command in the C- and I-style CLIs to use the compact representation for this path. The compact representation means that all leaf elements are shown on a single line.

Attributes:

src (pathType) — The "src" attribute is mandatory. It specifies which path to make compact. pathType is a space-separated list of elements, pointing out a specific element.

wrap (xs:boolean) — The "wrap" attribute is optional. It specified whether the line should be wrapped at screen-width or not.

delimiter (xs:string) — The "delimiter" attribute is optional. It specified which string to use between the element name and the value when displaying leaf values.

prettify(xs:boolean) — The "prettify" attribute is optional. If set to 'true' then dash:es and underscores will be replaced by spaces in leaf element names.

spacer (xs:string) — The "spacer" attribute is optional. It specified which string to use between the elements when displayed in compact format.

width (xs:positiveInteger) — The "width" attribute is optional. It specified a fixed terminal width to use before wrapping line. It is only used when wrap is set to 'true'. If width is not specified the line is wrapped when the terminal width is reached.

Note that the tailf:cli-compact-stats YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/columnWidth`

The "columnWidth" element can be used to set fixed widths for specific columns in auto-rendered tables.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to set the column width for. pathType

is a space-separated list of node names, pointing out a specific data model node.

width (xs:positiveInteger) — The "width" attribute is mandatory. It specified a fixed column width.

Note that the tailf:cli-column-width YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/columnAlign`

The "columnAlign" element can be used to specify the alignment of the data in specific columns in auto-rendered tables.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to set the column alignment for. pathType is a space-separated list of node names, pointing out a specific data model node.

align (left|right|center) — The "align" attribute is mandatory.

Note that the tailf:cli-column-align YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/defaultColumnAlign`

The "defaultColumnAlign" element can be used to specify a default alignment of a simpletype when used in auto-rendered tables.

Attributes:

namespace (xs:string) — The "namespace" attribute is required. It specifies in which namespace the type is found. It can be either the namespace URI or the namespace prefix.

name (xs:string) — The "name" attribute is required. It specifies the name of the type in the given namespace.

align (left|right|center) — The "align" attribute is mandatory.

`/clispec/$MODE/modifications/multiLinePrompt`

The "multiLinePrompt" element can be used to specify that the CLI should automatically enter multi-line prompt mode when prompting for values of the given type.

Attributes:

namespace (xs:string) — The "namespace" attribute is required. It specifies in which namespace the type is found. It can be either the namespace URI or the namespace prefix.

name (xs:string) — The "name" attribute is required. It specifies the name of the type in the given namespace.

`/clispec/$MODE/modifications/columnStats`

The "columnStats" element tells the Cisco style CLIs show command to display elements in a container as a column, ie to not repeat the name of the container element on each line but instead indent each leaf under the container.

Attributes:

src (pathType) — The "src" attribute is mandatory. It specifies which path to make display as a column. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-column-stats YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/showTemplate`

The "showTemplate" element is used for specifying a template to use by the show command in the J-, C- and I-style CLIs.

Show templates are associated with YANG nodes and are used by the CLI when the 'show' command is given for a path in operational mode. It is primarily intended for displaying "config false" data but "config true" data may be included in the template as well.

The template may contain a mix of text and expandable entries. Expandable entries all start with $(and end with a matching). Parentheses and dollar signes needs to be quoted in plain-text.

Expansion:

Parameter is either a relative or absolute path to a leaf (eg / foo/bar, foo/bar), or one of the builtin variables: .selected, .entered, .legend_shown, .user, .groups, .ip, .display_groups, .path, .ipath or .licounter. In addition the variables "spath" and "ispath" are available when a command is executed from a show path.

.selected — The *.selected* variable contains the list of selected paths to be shown. The show template can inspect this element to determine if a given element should be displayed or not. For example:

```
$(.selected~=hwaddr?HW Address)
```

.entered — The *.entered* variable is true if the "entered" text has been displayed (either the auto generated text or a showTemplateEnter). This is useful when having a non-table template where each instance should have a text.

```
$(.entered?:host $(name))
```

.legend_shown — The *.legend_shown* variable is true if the "legend" text has been displayed (either the auto generated table header or a showTemplateLegend). This is useful to inspect when displaying a table row. If the user enteres the path to a specific instance

the builtin table header will not be displayed and the showTemplateLegend will not be invoked and it may be useful to render the legend specifically for this instance.

```
$(.legend_shown!=true?Address         Interface)
```

.user The *.user* variable contains the name of the current user. This can be used for differentiating the content displayed for a specific user, or in paths. For example:

```
$(user{$(.user)}/settings)
```

.groups The *.groups* variable contains the a list of groups that the user belongs to.

.display_groups The *.display_groups* variable contains a list of selected display groups. This can be used to display different content depending on the selected display group. For example:

```
$(.display_groups~=details?details...)
```

.ip The *.ip* variable contains the ip address that the user connected from.

.path The *.path* variable contains the path to the entry, formatted in CLI style.

.ipath The *.ipath* variable contains the path to the entry, formatted in template style.

.spath The *.spath* variable contains the show path, formatted in CLI style.

.ispath The *.ipath* variable contains the show path, formatted in template style.

.licounter The *.licounter* variable contains a counter that is incremented for each instance in a list. This means that it will be 0 in the legend, contain the total number of list instances in the footer and something in between in the basic show template.

$(parameter)

The value of parameter is substituted.

$(cond?word1:word2)

The expansion of word1 is substituted if the value of cond evaluates to true, otherwise the expansion of word2 is substituted.

cond may be one of

parameter

Evaluates to true if the node exists.

parameter == <value>

Evaluates to true if the value of the parameter equals <value>

parameter != <value>

Evaluates to true if the value of the parameter does not equal <value>

parameter ~= <value>

Provided that parameters value is a list then this expression evaluates to true if <value> is a member of the list.

$(parameter|filter)

The value of parameter processed by filter is substituted. Filters may be either one of the built-ins or a customized filter defined in a callback. See /confdConfig/cli/templateFilter. A built-in filter may be one of

capfirst

Capitalizes the first character of the value.

filesizeformat

Format the value in a 'human-readable' format (i.e. '13 KB', '4.1 MB' '102 bytes' etc), where K means 1024, M means 1024*1024 etc.

When used without argument the default number of decimals displayed is 2. When used with a numeric integer argument, filesizeformat will display the given number of decimal places.

humanreadable

Similar to filesizeformat except no bytes suffix is added. (i.e. '13.13 k', '4.2 M' '102' etc), where k means 1000, M means 1000*1000 etc.

When used without argument the default number of decimals displayed is 2. When used with a numeric integer argument, humanreadable will display the given number of decimal places.

commasep

Separate the numerical values into groups of three digits using a comma (e.g., 1234567 -> 1,234,567)

hex

Display integer as hex number. An argument can be used to indicate how many digits should be used in the output. If the hex number is too long it will be truncated at the front, if it is too short it will be padded with zeros at the front. If the width is a negative number then at most that number of digits will be used, but short numbers will not be padded with zeroes. For example:

```
value       Template              Output
12345       {{ value|hex }}          3039
12345       {{ value|hex:2 }}         39
12345       {{ value|hex:8 }}       00003039
```

hexlist

Display integer as hex number with : between pairs. An argument can be used to indicate how many digits should be used in the output. If the hex number is too long it will be truncated at the front, if it is too short it will be padded with zeros at the front. If the width is a negative number then at most that number of digits will be used, but short numbers will not be padded with zeroes.For example:

```
value       Template              Output
12345       {{ value|hexlist }}      30:39
12345       {{ value|hexlist:2 }}     39
12345       {{ value|hexlist:8 }}     00:00:30:39
```

floatformat

Used for type 'float' in tailf-xsd-types. We recommend that the YANG built-in type 'decimal64' is used instead of 'float'.

When used without an argument, rounds a floating-point number to one decimal place -- but only if there's a decimal part to be displayed. For example:

```
value       Template                  Output
34.23234      {{ value|floatformat }}     34.2
34.00000      {{ value|floatformat }}     34
34.26000      {{ value|floatformat }}     34.3
```

If used with a numeric integer argument, floatformat rounds a number to that many decimal places. For example:

```
value       Template                  Output
34.23234      {{ value|floatformat:3 }}   34.232
34.00000      {{ value|floatformat:3 }}   34.000
```

```
34.26000      {{ value|floatformat:3 }}      34.260
```

If the argument passed to floatformat is negative, it will round a number to that many decimal places -- but only if there's a decimal part to be displayed. For example:

```
value          Template                    Output
34.23234       {{ value|floatformat:-3 }}  34.232
34.00000       {{ value|floatformat:-3 }}  34
34.26000       {{ value|floatformat:-3 }}  34.260
```

Using floatformat with no argument is equivalent to using floatformat with an argument of -1.

ljust:width

Left-align the value given a width.

rjust:width

Right-align the value given a width.

trunc:width

Truncate value to a given width.

lower

Convert the value into lowercase.

upper

Convert the value into uppercase.

show:<dictionary>

Substitutes the result of invoking the default display function for the parameter. The dictionary can be used for introducing own variables that can be accessed in the same manner as builtin variables. The user defined variables overrides builtin variables. The dictionary is specified as a string on the following form:

```
(key=value)(:key=value)*
```

For example, with the following expression:

```
$(foo|show:myvar1=true:myvar2=Interface)
```

the user defined variables can be accessed like this:

```
$(.myvar1!=true?Address) $(.myvar2)
```

dict:<dictionary>

Translate the value using the dictionary. Can for example be used for displaying on/off instead of true/false.

For example

```
$(foo|dict:true=on:false=off)
```

Nested invocations are allowed, ie it is possible to have expressions like $((state|dict:yes=Yes:no=No)|rjust:14), or $(/foo{$(../bar)}).

An example:

```
<modifications>
  <showTemplate path="interfaces">$(name) is administratively $(status), line protoco
  Hardware is $(hw), address is $(port_address) (bia $(bia))
  Internet address is $(address)
  MTU $(mtu) bytes, BW $(bw|humanreadable)bit, DLY $(dly) usec,
     reliability $(reliability), txload $(txload), rxload $(rxload)
  Encapsulation $(encapsulation|upper), $(loopback?:loopback not set)
  Keepalive $(keepalive?set to \($(keepalive) sec\):not set)
  ARP type: $(arpType), ARP Timeout $(arpTimeout)
  Last input $(lastInput), output $(output), output hang $(outputHang)
  Last clearing of "show interface" counters $(lastClear)
  Queuing strategy: $(queingSrategy)
  Output queue $(output/queue), $(output/drops) drops; input queue $(input/queue), $(
  5 minute input rate $(input/rate) bits/sec, $(input/packetRate) packets/sec
  5 minute output rate $(output/rate) bits/sec, $(output/packetRate) packets/sec
     $(input/packets) packets input, $(input/bytes) bytes, $(input/buffer) no buffer
     Received $(input/receivedBroadcasts) broadcasts, $(input/runts) runts, $(input/g
     $(input/errors) input errors, $(input/crc) CRC, $(input/frame) frame, $(input/ov
     $(input/dribble) input packets with dribble condition detected
     $(output/packets) packets output, $(output/bytes) bytes, $(output/underruns) und
     $(output/errors) output errors, $(output/collisions) collisions, $(output/reset
     $(output/babbles) babbles, $(output/lateCollision) late collision, $(output/defe
     $(lostCarrier) lost carrier, $(noCarrier) no carrier
     $(output/bufferFails) output buffer failures, $(output/bufferSwapped) output buf
</showTemplate>
</modifications>
```

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies on which path to apply the show template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-show-template YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/showTemplateLegend`

The "showTemplateLegend" element is used for specifying a template to use by the show command in the J-, C- and I-style CLIs when displaying a set of list nodes as a legend.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-show-template-legend YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/showTemplateEnter`

The "showTemplateEnter" element is used for specifying a template to use by the show command in the J-, C- and I-style CLIs when displaying a set of list element nodes before displaying each instance.

In addition to the builtin variables in ordinary templates there are two additional variables available: .prefix_str and .key_str.

.prefix_str The *.prefix_str* variable contains the text displayed before the key values when auto-rendering an enter text.

.key_str The *.key_str* variable contains the keys as a text

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-show-template-enter YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/showTemplateFooter`

The "showTemplateFooter" element is used for specifying a template to use by the show command in the J-, C- and I-style CLIs after a set of list nodes has been displayed as a table.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-show-template-footer YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/runTemplate`

The "run" element is used for specifying a template to use by the "show running-config" command in the C- and I-style CLIs. The syntax is the same as for the showTemplate above. The template is only used if it is associated with a leaf element. Containers and lists cannot have runTemplates.

Note that extreme care must be taken when using this feature if the result should be paste:able into the CLI again.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show running-config template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-run-template YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/runTemplateLegend`

The "runTemplateLegend" element is used for specifying a template to use by the show running-config command in the C- and I-style CLIs when displaying a set of list nodes as a legend.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show running-config template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-run-template-legend YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/runTemplateEnter`

The "runTemplateEnter" element is used for specifying a template to use by the show running-config command in the C- and I-style CLIs when displaying a set of list element nodes before displaying each instance.

In addition to the builtin variables in ordinary templates there are two additional variables available: .prefix_str and .key_str.

.prefix_str The *.prefix_str* variable contains the text displayed before the key values when auto-rendering an enter text.

.key_str The *.key_str* variable contains the keys as a text

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show running-config template. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-run-template-enter YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/runTemplateFooter`

The "runTemplateFooter" element is used for specifying a template to use by the show running-config command in the C- and I-style CLIs after a set of list nodes has been displayed as a table.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to apply the show running-config template. pathType is a space-

separated list of elements, pointing out a specific container element.

Note that the tailf:cli-run-template-footer YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/hasRange`

The "hasRange" element is used for specifying that a given non-integer key element should allow range expressions

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies on which path to allow range expressions. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-allow-range YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressRange`

The "suppressRange" element is used for specifying that a given integer key element should not allow range expressions

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies on which path to suppress range expressions. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-range YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/customRange`

The "customRange" element is used for specifying that a given list element should support ranges. A type matching the range expression must be supplied, as well as a callback to use to determine if a given instance is covered by a given range expression. It contains one or more "rangeType" elements and one "callback" element.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies on which path to apply the custom range. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-custom-range YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/customRange/callback`

The "callback" element is used for specifying which callback to invoke for checking if a list element instance belongs to a range. It contains a "capi" element.

Note that the tailf:cli-custom-range-actionpoint YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/customRange/callback/capi`

The "capi" element is used for specifying the name of the callback to invoke for checking if a list element instance belongs to a range.

Attributes:

id (string) The "id" attribute is optional. It specifies a string which is passed to the callback when invoked to check if a value belongs in a range. This makes it possible to use the same callback at several locations and still keep track of which point it is invoked from.

`/clispec/$MODE/modifications/customRange/rangeType`

The "rangeType" element is used for specifying which key element of a list element should support range expressions. It is also used for specifying a matching type. All range expressions must belong to the specified type, and a valid key element must not be a valid element of this type.

Attributes:

key (string) The "key" attribute is mandatory. It specifies which key element of the list that the rangeType applies to.

namespace (string) The "namespace" attribute is mandatory. It specifies which namespace the type belongs to.

name (string) The "name" attribute is mandatory. It specifies the name of the range type.

Note that the tailf:cli-range-type YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/allowWildcard`

The "allowWildcard" element is used for specifying that a given list element should allow wildcard expressions in the show pattern

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to allow wildcard expressions. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-allow-wildcard YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressWildcard`

The "suppressWildcard" element is used for specifying that a given list element should not allow wildcard expressions in the show pattern

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies on which path to suppress wildcard expressions. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-wildcard YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressValidationWarningPrompt`

The "suppressValidationWarningPrompt" element is used for specifying that for a given path a validate warning should not result in a prompt to the user. The warning is displayed but without blocking the commit operation.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies on which path to suppress the validation warning prompt. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-validate-warning-prompt YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/errorMessageRewrite`

The "errorMessageRewrite" element is used for specifying that a callback should be invoked for possibly rewriting error messages before displaying them.

`/clispec/$MODE/modifications/errorMessageRewrite/callback`

The "callback" element is used for specifying which callback to invoke for rewriting a message. It contains a "capi" element.

`/clispec/$MODE/modifications/errorMessageRewrite/callback/capi`

The "capi" element is used for specifying the name of the callback to invoke for rewriting a message.

`/clispec/$MODE/modifications/showPathRewrite`

The "showPathRewrite" element is used for specifying that a callback should be invoked for possibly rewriting the show path before executing a show command. The callback is invoked by the builtin show command.

`/clispec/$MODE/modifications/showPathRewrite/callback`

The "callback" element is used for specifying which callback to invoke for rewriting the show path. It contains a "capi" element.

`/clispec/$MODE/modifications/showPathRewrite/callback/capi`

The "capi" element is used for specifying the name of the callback to invoke for rewriting the show path.

`/clispec/$MODE/modifications/noKeyCompletion`

The "noKeyCompletion" element tells the CLI to not perform completion for key elements for a given path. This is to avoid querying the data provider for all existing keys.

Attributes:

src (pathType) — The "src" attribute is mandatory. It specifies which path to make not do completion for. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-no-key-completion extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/noMatchCompletion`

The "noMatchCompletion" element tells the CLI to not provide match completion for a given element path for show commands.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to make not do match completion for. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-no-match-completion YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/incompleteShowPath`

The "incompleteShowPath" element makes it possible to specify that a path to the show command in the C and I-style CLIs is considered incomplete. It can also be used to specify that a minimum number of keys needs to be specified.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to consider incomplete. pathType is a space-separated list of elements, pointing out a specific list element.

minKeys (positiveInteger) — The "minKeys" attribute is optional. For paths leading to a list element it is possible to specify the minimum number of required keys.

Note that the tailf:cli-incomplete-show-path YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressShowPath`

The "suppressShowPath" element makes it possible to specify that a path to the show command should not be available. This only applies to I- and C- style CLI.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-show-path YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressShowMatch`

The "suppressShowMatch" element makes it possible to specify that a specific completion match (ie a filter match that appear at list element nodes as an alternative to specifying a single instance) to the show command should not be available.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-show-match YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/incompleteCommand`

The "incompleteCommand" element makes it possible to specify that an auto-rendered command in C- and I-mode should be considered incomplete. It can be used to prevent <cr> from appearing in the completion list for optional internal nodes.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies which path to consider incomplete. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-incomplete-command YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/sequenceCommands`

The "sequenceCommands" element makes it possible to specify that an auto-rendered command in C- and I-mode should only accept arguments in the same order as they are specified in the YANG module. This, in combination with dropElem, can be used to create CLI commands for setting multiple leafs in a container without having to specify the leaf names.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies which path to make into an argument sequence. pathType is a space-separated list of elements, pointing out a specific container element.

Note that the tailf:cli-sequence-commands YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/fullCommand`

The "fullCommand" element makes it possible to specify that an auto-rendered command in C- and I-mode should be considered full. It can be used to prevent further command stacking.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies which path to consider full. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-full-command YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/fullShowPath`

The "fullShowPath" element makes it possible to specify that a path to the show command in the C and I-style CLIs is considered complete. No further elements can be added to the path.

Attributes:

path (pathType)	The "path" attribute is mandatory. It specifies which path to consider complete. pathType is a space-separated list of elements, pointing out a specific list element.
maxKeys (positiveInteger)	The "maxKeys" attribute is optional. For paths leading to a list element it is possible to specify the maximum number of allowed keys.

Note that the tailf:cli-full-show-path YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/multiValue`

The "multiValue" element tells the parser for the C- and I- style CLIs that a specific leaf element should get its value from the rest of the command line. If this modification is used for a given leaf it will not be possible to enter any more leaf values on the same command line.

Attributes:

src (pathType)	The "src" attribute is mandatory. It specifies which path to make a multiline input item by default. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-multi-value YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/enforceTable`

The "enforceTable" element makes it possible to force the generation of a table for a list element node regardless of whether the table will be too wide or not. This applies to the tables generated by the auto-rendered show commands for config="false" data in the C- and I- style CLIs.

Attributes:

src (pathType)	The "src" attribute is mandatory. It specifies which path to enforce. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-enforce-table YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressTable`

The "suppressTable" element makes it possible to suppress an automatically generated table in the C- and I- style CLIs when using the show command.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-table YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/preformatted`

The "preformatted" element makes it possible to suppress quoting of stats elements when displaying them. Newlines will be preserved in strings etc

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to consider preformatted. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-preformatted YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/exposeKeyName`

The "exposeKeyName" element makes it possible to force the C- and I-style CLIs to expose the key name to the CLI user. The user will be required to enter the name of the key and the key name will be displayed when showing the configuration.

Attributes:

path (pathType) The "src" attribute is mandatory. It specifies which leaf to expose. pathType is a space-separated list of elements, pointing out a specific list key element.

Note that the tailf:cli-expose-key-name YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/displayEmptyConfig`

The "displayEmptyConfig" element makes it possible to tell confd to display empty configuration list elements when displaying stats data in J-style CLI, provided that the list element has at least one optional config="false" element.

Attributes:

path (pathType) The "path" attribute is mandatory. It specifies which path to apply the mod to. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-display-empty-config YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressKeyAbbrev`

The "suppressKeyAbbrev" element makes it possible to suppress the use of abbreviations for specific key elements.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-key-abbreviation YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/allowKeyAbbrev`

The "allowKeyAbbrev" element makes it possible to allow the use of abbreviations for specific key elements.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:allow-key-abbreviation YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/suppressMode`

The "suppressMode" element makes it possible to suppress an automatically generated mode in C- and I- style CLI.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies which path to suppress. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-mode YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/addMode`

The "addMode" element makes it possible to create a mode at a non-list element. Only applicable in C- and I- style CLI.

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies for which path to create the mode. pathType is a space-separated list of elements, pointing out a specific non-list, non-leaf element.

Note that the tailf:cli-add-mode YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/modeName`

The "modeName" element makes it possible to specify a custom mode name in the C- and I- style CLI. It contains one of the elements "fixed" or "capi".

Attributes:

src (pathType) The "src" attribute is mandatory. It specifies for which path the custom mode name should apply. pathType is a space-separated list of elements, pointing out a path to a mode.

`/clispec/$MODE/modifications/modeName/fixed (xs:string)`

Specifies a fixed mode name.

Note that the tailf:cli-mode-name YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/modeName/capi`

Specifies that the mode name should be calculated through a callback function. It contains exactly one "cmdpoint" element.

Note that the tailf:cli-mode-name-actionpoint YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/modeName/capi/cmdpoint (xs:string)`

Specifies the callpoint name of the mode name function.

`/clispec/$MODE/modifications/autocommitDelay`

The "autocommitDelay" element makes it possible to enable transactions while in a specific submode (or submode of that mode). The modifications performed in that mode will not take effect until the user exits that submode.

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to delay autocommit for. pathType is a space-separated list of elements, pointing out a specific non-list, non-leaf element.

Note that the tailf:cli-delayed-auto-commit YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/simpleType`

The "simpleType" element makes it possible to customize the handling of a type. A type is typically attached to each leaf in a YANG module or to command parameters. The "simpleType" element makes it possible to customize both builtin and derived types. For now the only handling that can be customized is how CLI completion is to be performed.

The "simpleType" element contains a single element "capi".

Attributes:

namespace (string) — The "namespace" attribute is mandatory. It specifies the namespace URI whereas the type to be modified has been defined.

name (string) — The "name" attribute is mandatory. It specifies the name of the type to be customized

`/clispec/$MODE/modifications/simpleType/capi`

Specifies that the simpleType customization should be calculated through a callback function. It contains exactly one "completionpoint" element.

`/clispec/$MODE/modifications/simpleType/capi/completionpoint (xs:string)`

Specifies the callpoint name of the completion function.

`/clispec/$MODE/modifications/completion`

The "completion" element makes it possible to customize the completion of a specific path.

The "completion" element contains a single element "capi" enclosed in the "callback" element.

Attributes:

path (cmdpathType) — The "path" attribute is mandatory. It specifies for which path the completion callback should be applied to. cmdpathType is a space-separated list of commands.

Note that the tailf:cli-completion-actionpoint YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/completion/callback/capi`

Specifies that the completion customization should be calculated through a callback function. It contains exactly one "completionpoint" element.

`/clispec/$MODE/modifications/completion/callback/capi/completionpoint (xs:string)`

Specifies the callpoint name of the completion function.

Attributes:

id (string) — The "id" attribute is optional. It specifies a string which is passed to the callback when invoked. This makes it possible to use the same callback at several locations and still keep track of which point it is invoked from.

`/clispec/$MODE/modifications/suppressKeySort`

The "suppressKeySort" element makes it possible to suppress sorting of key-values in the completion list. Instead the values will be displayed in the same order as they are provided by the data-provider (external or CDB).

Attributes:

path (pathType) — The "path" attribute is mandatory. It specifies which path to not sort. pathType is a space-separated list of elements, pointing out a specific list element.

Note that the tailf:cli-suppress-key-sort YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/legend` (xs:string)

The "legend" element makes it possible to add a custom legend to be displayed when before printing a table. The legend is specified as a template string.

Attributes:

path (cmdpathType) — The "path" attribute is mandatory. It specifies for which path the legend should be printed. cmdpathType is a space-separated list of commands.

Note that the tailf:cli-legend YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/footer` (xs:string)

The "footer" element makes it possible to specify a template that will be displayed after printing a table.

Attributes:

path (cmdpathType) — The "path" attribute is mandatory. It specifies for which path the footer should be printed. cmdpathType is a space-separated list of commands.

Note that the tailf:cli-footer YANG extension can be used to the same effect directly in YANG file.

`/clispec/$MODE/modifications/help` (xs:string)

The "help" element makes it possible to add a custom help text to the specified built-in command.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to add the text to. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/paramhelp` (xs:string)

The "paramhelp" element makes it possible to add a custom help text to a parameter to a specified built-in command.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to add the text to. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

nr (positiveInteger) — The "nr" attribute is mandatory. It specifies which parameter of the command to add the text to.

`/clispec/$MODE/modifications/typehelp` (xs:string)

The "typehelp" element makes it possible to add a custom help text for the built-in primitive types, e.g. to change the default type name in the CLI. For example, to display "<integer>" instead of "<unsignedShort>".

The built-in primitive types are: string, atom, normalizedString, boolean, float, decimal, double, hexBinary, base64Binary, anyURI, anySimpleType, QName, NOTATION, token, integer, nonPositiveInteger, negativeInteger, long, int, short, byte, nonNegativeInteger, unsignedLong, positiveInteger, unsignedInt, unsignedShort, unsignedByte, dateTime, date, gYearMonth, gDay, gYear, time, gMonthDay,

gMonth, duration, inetAddress, inetAddressIPv4, inetAddressIP, inetAddressIPv6, inetAddressDNS, inetPortNumber, size, MD5DigestString, DES3CBCEncryptedString, AESCFB128EncryptedString, objectRef, bits_type_32, bits_type_64, hexValue, hexList, octetList, Gauge32, Counter32, Counter64, and oid.

Attributes:

type (xs:Name) The "type" attribute is mandatory. It specifies which primitive type to modify.

`/clispec/$MODE/modifications/info` (xs:string)

The "info" element makes it possible to add a custom info text to the specified built-in command.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to hide. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/paraminfo` (xs:string)

The "paraminfo" element makes it possible to add a custom info text to a parameter to a specified built-in command.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to add the text to. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

nr (positiveInteger) The "nr" attribute is mandatory. It specifies which parameter of the command to add the text to.

`/clispec/$MODE/modifications/timeout` (xs:integer|infinity)

The "timeout" element makes it possible to add a custom command timeout (in seconds) to the specified built-in command.

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to add the timeout to. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

`/clispec/$MODE/modifications/hide`

The "hide" element makes it possible to hide a built-in command

Attributes:

src (cmdpathType) The "src" attribute is mandatory. It specifies which command to hide. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

An example:

```
<modifications>
<hide src="file show"/>
</modifications>
```

`/clispec/$MODE/modifications/hideGroup`

The "hideGroup" element makes it possible to hide a built-in command under a hide group.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to hide. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

name (xs:string) — The "name" attribute is mandatory. It specifies which hide group to hide the command.

An example:

```
<modifications>
  <hideGroup src="file show" name="debug"/>
</modifications>
```

`/clispec/$MODE/modifications/submodeCommand`

The "submodeCommand" element makes it possible to make a command visible in the completion lists of all submodes.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to make available. cmdpathType is a space-separated list of commands, pointing out a specific sub-command.

An example:

```
<modifications>
  <submodeCommand src="clear"/>
</modifications>
```

`/clispec/$MODE/modifications/confirmText` (xs:string)

The "confirmText" element makes it possible to add a confirmation text to the specified command, i.e. the CLI user is prompted whenever this command is executed. The prompt to be used is given as a body to the element as seen in confd-light.cli above. The valid answers are "yes" and "no" - the text " [yes, no]" will automatically be added to the given confirmation text.

Attributes:

src (cmdpathType) — The "src" attribute is mandatory. It specifies which command to add a confirmation prompt to. cmd-

pathType is a space-separated list of commands, pointing out a specific sub-command.

defaultOption (yes|no) — The "defaultOption" attribute is optional. It makes it possible to customize if "yes" or "no" should be the default option, i.e. if the user just hits ENTER. If this element is not defined it defaults to whatever is specified by the /clispec/$MODE/modifications/defaultConfirmOption element.

`/clispec/$MODE/modifications/defaultConfirmOption` (yes|no)

The "defaultConfirmOption" element makes it possible to customize if "yes" or "no" should be the default option, i.e. if the user just hits ENTER, for the confirmation text added by the "confirmText" element.

If this element is not defined it defaults to "yes".

This element affects both /clispec/$MODE/modifications/confirmText and /clispec/$MODE/cmd/confirmText if they have not defined their "defaultOption" attributes.

`/clispec/$MODE/modifications/keymap`

The "keymap" element makes it possible to modify the key bindings in the command line editor.

Attributes:

key (xs:string) — The "key" attribute is mandatory. It specifies which sequence of keystrokes to modify.

action (keymapActionType) — The "action" attribute is mandatory. It specifies what should happen when the specified key sequence is executed. Possible values are: "unset", "new", "exist", "start_of_line", "back", "abort", "tab", "delete_forward", "delete_forward_no_eof", "end_of_line", "forward", "kill_rest", "redraw", "redraw_clear", "newline", "insert(chars)", "history_next", "history_prev", "isearch_back", "transpose", "kill_line", "quote", "word_delete_back", "yank", "end_mode", "delete", "word_delete_forward", "beginning_of_line", "delete", "end_of_line", "word_forward", "word_back", "end_of_line", "beginning_of_line", "word_back", "word_forward", "word_capitalize", "word_lowercase", "word_uppercase", "word_delete_back", "word_delete_forward", "multiline_mode", "yank_killring", and "quot". To remove a default binding use the action "remove_binding".

The default keymap is:

```
      <keymap key="\^A" action="start_of_line"/>
```

```
      <keymap key="\^B" action="back"/>
      <keymap key="\^C" action="abort"/>
      <keymap key="\^D" action="delete_forward"/>
      <keymap key="\^E" action="end_of_line"/>
      <keymap key="\^F" action="forward"/>
      <keymap key="\^J" action="newline"/>
      <keymap key="\^K" action="kill_rest"/>
      <keymap key="\^L" action="redraw_clear"/>
      <keymap key="\^M" action="newline"/>
      <keymap key="\^N" action="history_next"/>
      <keymap key="\^P" action="history_prev"/>
      <keymap key="\^R" action="isearch_back"/>
      <keymap key="\^T" action="transpose"/>
      <keymap key="\^U" action="kill_line"/>
      <keymap key="\^V" action="quote"/>
      <keymap key="\^W" action="word_delete_back"/>
      <keymap key="\^X" action="kill_line"/>
      <keymap key="\^Y" action="yank"/>
      <keymap key="\^Z" action="end_mode"/>
      <keymap key="\d" action="delete"/>
      <keymap key="\t" action="tab"/>
      <keymap key="\b" action="delete"/>
      <keymap key="\ed" action="word_delete_forward"/>
      <keymap key="\e[Z" action="tab"/>
      <keymap key="\e[A" action="history_prev"/>
      <keymap key="\e[1~" action="beginning_of_line"/>
      <keymap key="\e[3~" action="delete"/>
      <keymap key="\e[4~" action="end_of_line"/>
      <keymap key="\eOA" action="history_prev"/>
      <keymap key="\eOB" action="history_next"/>
      <keymap key="\eOC" action="forward"/>
      <keymap key="\eOD" action="back"/>
      <keymap key="\eOM" action="newline"/>
      <keymap key="\eOp" action="insert(0)"/>
      <keymap key="\eOq" action="insert(1)"/>
      <keymap key="\eOr" action="insert(2)"/>
      <keymap key="\eOs" action="insert(3)"/>
      <keymap key="\eOt" action="insert(4)"/>
      <keymap key="\eOu" action="insert(5)"/>
      <keymap key="\eOv" action="insert(6)"/>
      <keymap key="\eOw" action="insert(7)"/>
      <keymap key="\eOx" action="insert(8)"/>
      <keymap key="\eOy" action="insert(9)"/>
      <keymap key="\eOm" action="insert(-)"/>
      <keymap key="\eOl" action="insert(*)"/>
      <keymap key="\eOn" action="insert(.)"/>
      <keymap key="\e[5C" action="word_forward"/>
      <keymap key="\e[5D" action="word_back"/>
      <keymap key="\e[1;5C" action="word_forward"/>
      <keymap key="\e[1;5D" action="word_back"/>
      <keymap key="\e[B" action="history_next"/>
      <keymap key="\e[C" action="forward"/>
      <keymap key="\e[D" action="back"/>
      <keymap key="\e[F" action="end_of_line"/>
      <keymap key="\e[H" action="beginning_of_line"/>
      <keymap key="\eb" action="word_back"/>
      <keymap key="\ef" action="word_forward"/>
      <keymap key="\ec" action="word_capitalize"/>
      <keymap key="\el" action="word_lowercase"/>
      <keymap key="\eu" action="word_uppercase"/>
```

```
      <keymap key="\e\b" action="word_delete_back"/>
      <keymap key="\e\d" action="word_delete_back"/>
      <keymap key="\ed" action="word_delete_forward"/>
      <keymap key="\em" action="multiline_mode"/>
      <keymap key="\ey" action="yank_killring"/>
      <keymap key="\eq" action="quote"/>
```

The default keymap for I-style differs with the following mapping:

```
  <keymap key="\^D" action="delete_forward_no_eof"/>
```

`/clispec/$MODE/show`

The "show" element overrides the built-in show command, in the C-style CLI, for a givenpath defined by the "path" attribute. It contains (in order) zero or one "callback" elements, and zero or one "options" elements.

Attributes:

path (showpathType) [] The "path" attribute is required. It specifies for which path the command should be invoked.

An example:

```
<show path="aaa authentication users user">
  <callback>
    <exec>
      <osCommand>./show_aaa_auth.sh</osCommand>
    </exec>
  </callback>
</show>
```

`/clispec/$MODE/show/callback`

The "callback" element specifies how the command is implemented, e.g. as a OS executable or a CAPI callback. It contains one of the elements "capi", and "exec".

`/clispec/$MODE/show/callback/mode`

The "mode" element specifies that the command is used for entering a custom mode. It contains one "name" and one "datastore" element.

An example:

```
<callback>
  <mode>
    <name>debug</name>
    <datastore>private</name>
  </mode>
</callback>
```

`/clispec/$MODE/show/callback/mode/name` (xs:NCName)

The "name" element specifies the name of the custom mode. For this to work, a custom mode with that name must be declared.

`/clispec/$MODE/show/callback/mode/datastore`*[private]*

The "datastore" element must be one of "private", "shared" and "exclusive". It is ignored for operational custom modes and when entering a configure mode from within another configure mode. It is only used when going from operational mode to configure mode.

`/clispec/$MODE/show/callback/capi`

The "capi" element specifies that the command is implemented using C-API using the same API as for actions. It contains one "cmdpoint" element and one or zero "args" element.

An example:

```
<callback>
  <capi>
    <cmdpoint>adduser</cmdpoint>
  </capi>
</callback>
```

`/clispec/$MODE/show/callback/capi/args` (argsType)

The "args" element specifies the arguments to use when executing the command specified by the "callpoint" element. argsType is a space-separated list of argument strings.

The string may contain a number of built-in variables which are expanded on execution. The built-in variables are: "cwd", "user", "groups", "ip", "maapi", "uid", "gid", "tty", "ssh_connection", "opaque", "path", "cpath", "ipath" and "licounter". In addition the variables "spath" and "ispath" are available when a command is executed from a show path. For example:

```
<args>$(user)</args>
```

Will expand to the username.

`/clispec/$MODE/show/callback/capi/cmdpoint` (xs:NCName)

The "cmdpoint" element specifies the name of the C-API action to be called. For this to work, a actionpoint must be registered with the ConfD daemon at startup.

`/clispec/$MODE/show/callback/exec`

The "exec" element specifies how the command is implemented using an executable or a shell script. It contains (in order) one "osCommand" element, zero or one "args" elements and zero or one "options" elements.

An example:

```
<callback>
  <exec>
    <osCommand>cp</osCommand>
    <options>
      <uid><phrase condition="confd">confd</phrase><phrase condition="ncs">ncs</phras
      <wd>/var/tmp</wd>
      ...
    </options>
  </exec>
```

```
</callback>
```

`/clispec/$MODE/show/callback/exec/osCommand` (xs:token)

The "osCommand" element specifies the path to the executable or shell script to be called. If the command is in the $PATH (as specified when we start the ConfD daemon) the path may just be the name of the command.

The "osCommand" and "args" for "show" differs a bit from the ones for "cmd". For "show" there are a few built-in arguments that always are given to the "osCommand". These are appended to "args". The built-in arguments are "0", the keypath (ispath) and an optional filter. Like this: "0 /prefix:keypath *".

The command is invoked as if it had been executed by exec(3), i.e. not in a shell environment such as "/bin/sh -c ...".

`/clispec/$MODE/show/callback/exec/args` (argsType)

The "args" element specifies additional arguments to use when executing the command specified by the "osCommand" element. The "args" arguments are prepended to the mandatory ones listed in "osCommand". argsType is a space-separated list of argument strings.

The string may contain a number of built-in variables which are expanded on execution. The built-in variables are: "cwd", "user", "groups", "ip", "maapi", "uid", "gid", "tty", "ssh_connection", "opaque", "path", "cpath", "ipath" and "licounter". In addition the variables "spath" and "ispath" are available when a command is executed from a show path. For example:

```
<args>$(user)</args>
```

Will expand to the username and the three built-in arguments. For example: "admin 0 / prefix:keypath *".

`/clispec/$MODE/show/callback/exec/options`

The "options" element specifies how the command is be executed. It contains (in any order) zero or one "uid" elements, zero or one "gid" elements, zero or one "wd" elements, zero or one "batch" elements, zero or one "pty" element, zero or one of "interrupt" elements, zero or one of "noInput", and zero or one "ignoreExitValue" elements.

`/clispec/$MODE/show/callback/exec/options/uid` (idType) [confd]

The "uid" element specifies which user id to use when executing the command. Possible values are:

confd (default)	The command is run as the same user id as the ConfD daemon.
user	The command is run as the same user id as the user logged in to the CLI, i.e. we have to make sure that this user id exists as an actual user id on the device.
root	The command is run as root.

<uid> (the numerical user *<uid>*)	The command is run as the user id <uid>. *Note:* If uid is set to either "user", "root" or "<uid>" the the ConfD daemon must have been started as root (or setuid), or the showptywrapper must have setuid root permissions.

`/clispec/$MODE/show/callback/exec/options/gid` (idType) [confd]

The "gid" element specifies which group id to use when executing the command. Possible values are:

confd (default)	The command is run as the same group id as the ConfD daemon.
user	The command is run as the same group id as the user logged in to the CLI, i.e. we have to make sure that this group id exists as an actual group on the device.
root	The command is run as root.
<gid> (the numerical group *<gid>*)	The command is run as the group id <gid>. *Note:* If gid is set to either "user", "root" or "<gid>" the the ConfD daemon must have been started as root (or setuid), or the showptywrapper must have setuid root permissions.

`/clispec/$MODE/show/callback/exec/options/wd` (xs:token)

The "wd" element specifies which working directory to use when executing the command. If not given, the command is executed from the location of the CLI.

`/clispec/$MODE/show/callback/exec/options/pty` (xs:boolean)

The "pty" element specifies weather a pty should be allocated when executing the command. The default is to allocate a pty for operational and configure osCommands, but not for osCommands executing as a pipe command. This behavior can be overridden with this parameter.

`/clispec/$MODE/show/callback/exec/options/interrupt` (interruptType) [sigkill]

The "interrupt" element specifies what should happen when the user enters ctrl-c in the CLI. Possible values are:

sigkill (default)	The command is terminated by sending the sigkill signal.
sigint	The command is interrupted by the sigint signal.
sigterm	The command is interrupted by the sigterm signal.
ctrlc	The command is sent the ctrl-c character which is interpreted by the pty.

`/clispec/$MODE/show/callback/exec/options/ignoreExitValue`

The "ignoreExitValue" element specifies that the CLI engine should ignore the fact that the command returns a non-zero value. Normally it signals an error on stdout if a non-zero value is returned.

`/clispec/$MODE/show/callback/exec/options/globalNoDuplicate` (xs:token)

The "globalNoDuplicate" element specifies that only one instance with the same name can be run at any one time in the system. The command can be started either from the CLI, the Web UI or through NETCONF.

`/clispec/$MODE/show/callback/exec/options/noInput` (xs:token)

The "noInput" element specifies that the command should not grab the input stream and consume freely from that. This option should be used if the command should not consume input characters. If not used then the command will eath all data from the input stream and cut-and-paste may not work as intended.

`/clispec/$MODE/show/options`

The "options" element specifies under what circumstances the CLI command should execute. It contains (in any order) zero or one "notInterruptible" elements, zero or one of "displayWhen" elements, and zero or one "paginate" elements.

`/clispec/$MODE/show/options/notInterruptible`

The "notInterruptible" element disables <ctrl-c> and the execution of the CLI command can thus not be interrupted.

`/clispec/$MODE/show/options/paginate`

The "paginate" element enables a filter for paging through CLI command output text one screen at a time.

`/clispec/$MODE/show/options/displayWhen`

The "displayWhen" element can be used to add a displayWhen xpath condition to a command.

Attributes:

exrp (xpath expression)	The "expr" attribute is mandatory. It specifies an xpath expression. If the expression evaluates to true then the command is available, otherwise not.
ctx (path)	The "ctx" attribute is optional. If not specified the current editpath/mode-path is used as context node for the xpath evaluation. Note that the xpath expression will automatically evaluate to false if a display when expression is used for a top-level command and no ctx is specified. The path may contain variables defined in the dict.

`/clispec/operationalMode/start`

The "start" command is executed when the CLI is started. It can be used to, for example, remind the user to change an expired password. It contains (in order) zero or one "callback" elements, and zero or one "options" elements.

This element must occur after the <modifications> section and before any <cmd> entries.

An example:

```
<start>
  <callback>
    <exec>
      <osCommand>./startup.sh</osCommand>
    </exec>
  </callback>
</start>
```

`/clispec/operationalMode/start/callback`

The "callback" element specifies how the command is implemented, e.g. as a OS executable or an API callback. It contains one of the elements "capi", and "exec".

`/clispec/operationalMode/start/callback/capi`

The "capi" element specifies that the command is implemented using C-API using the same API as for actions. It contains one "cmdpoint" element.

An example:

```
<callback>
  <capi>
    <cmdpoint>adduser</cmdpoint>
  </capi>
</callback>
```

`/clispec/operationalMode/start/callback/capi/cmdpoint` (xs:NCName)

The "cmdpoint" element specifies the name of the C-API action to be called. For this to work, a actionpoint must be registered with the ConfD daemon at startup.

`/clispec/operationalMode/start/callback/exec`

The "exec" element specifies how the command is implemented using an executable or a shell script. It contains (in order) one "osCommand" element, zero or one "args" elements and zero or one "options" elements.

An example:

```
<callback>
  <exec>
    <osCommand>cp</osCommand>
    <options>
      <uid>confd</uid>
      <wd>/var/tmp</wd>
      ...
    </options>
  </exec>
</callback>
```

`/clispec/operationalMode/start/callback/exec/osCommand` (xs:token)

The "osCommand" element specifies the path to the executable or shell script to be called. If the command is in the $PATH (as specified when we start the ConfD daemon) the path may just be the name of the command.

The command is invoked as if it had been executed by exec(3), i.e. not in a shell environment such as "/bin/sh -c ...".

`/clispec/operationalMode/start/callback/exec/args` (argsType)

The "args" element specifies the arguments to use when executing the command specified by the "osCommand" element. argsType is a space-separated list of argument strings. The built-in variables are: "cwd", "user", "groups", "ip", "maapi", "uid", "gid", "tty", "ssh_connection", "opaque", "path", "cpath", "ipath" and "licounter". In addition the variables "spath" and "ispath" are available when a command is executed from a show path. For example:

```
<args>$(user)</args>
```

Will expand to the username.

`/clispec/operationalMode/start/callback/exec/options`

The "options" element specifies how the command is be executed. It contains (in any order) zero or one "uid" elements, zero or one "gid" elements, zero or one "wd" elements, zero or one "batch" elements, zero or one of "interrupt" elements, and zero or one "ignoreExitValue" elements.

`/clispec/operationalMode/start/callback/exec/options/uid` (idType) [confd]

The "uid" element specifies which user id to use when executing the command. Possible values are:

confd (default)	The command is run as the same user id as the ConfD daemon.
user	The command is run as the same user id as the user logged in to the CLI, i.e. we have to make sure that this user id exists as an actual user id on the device.
root	The command is run as root.
<*uid*> (the numerical user <*uid*>)	The command is run as the user id <uid>. *Note:* If uid is set to either "user", "root" or "<uid>" the the ConfD daemon must have been started as root (or setuid), or the startptywrapper must have setuid root permissions.

`/clispec/operationalMode/start/callback/exec/options/gid` (idType) [confd]

The "gid" element specifies which group id to use when executing the command. Possible values are:

confd (default)	The command is run as the same group id as the ConfD daemon.

user	The command is run as the same group id as the user logged in to the CLI, i.e. we have to make sure that this group id exists as an actual group on the device.
root	The command is run as root.
<*gid*> (the numerical group <*gid*>)	The command is run as the group id <gid>. *Note:* If gid is set to either "user", "root" or "<gid>" the the ConfD daemon must have been started as root (or setuid), or the startptywrapper must have setuid root permissions.

`/clispec/operationalMode/start/callback/exec/options/wd` (xs:token)

The "wd" element specifies which working directory to use when executing the command. If not given, the command is executed from the location of the CLI.

`/clispec/operationalMode/start/callback/exec/options/globalNoDuplicate` (xs:token)

The "globalNoDuplicate" element specifies that only one instance with the same name can be run at any one time in the system. The command can be started either from the CLI, the Web UI or through NETCONF.

`/clispec/operationalMode/start/callback/exec/options/interrupt` (interruptType) [sigkill]

The "interrupt" element specifies what should happen when the user enters ctrl-c in the CLI. Possible values are:

sigkill (default)	The command is terminated by sending the sigkill signal.
sigint	The command is interrupted by the sigint signal.
sigterm	The command is interrupted by the sigterm signal.
ctrlc	The command is sent the ctrl-c character which is interpreted by the pty.

`/clispec/operationalMode/start/callback/exec/options/ignoreExitValue`(xs:boolean) [false]

The "ignoreExitValue" element specifies if the CLI engine should ignore the fact that the command returns a non-zero value. Normally it signals an error on stdout if a non-zero value is returned.

`/clispec/operationalMode/start/options`

The "options" element specifies under what circumstances the CLI command should execute. It contains (in any order) zero or one "notInterruptible" elements, and zero or one "paginate" elements.

`/clispec/operationalMode/start/options/notInterruptible`

The "notInterruptible" element disables <ctrl-c> and the execution of the CLI command can thus not be interrupted.

`/clispec/operationalMode/start/options/paginate`

The "paginate" element enables a filter for paging through CLI command output text one screen at a time.

`/clispec/$MODE/cmd`

The "cmd" element adds a new command to the CLI hierarchy as defined by its "mount" and "mode" attributes. It contains (in order) one "info" element, one "help" element, zero or one "confirmText" element, zero or one "callback" elements, zero or one "params" elements, zero or one "options" elements and finally zero or more "cmd" elements (recursively).

Attributes:

name (xs:NCName) — The "name" attribute is mandatory. It specifies the name of the command.

mode (cmdpathType) [] — The "mode" attribute is optional. It specifies that the command should be mounted in a specific submode. The attribute is only applicable in the C- and I-style CLIs. If no "mode" attribute is given the command is mounted in the topmost mode.

extend (xs:boolean) [false] — The "extend" attribute is optional. It specifies that the command should be mounted on top of an existing command, ie with the exact same name as an existing command but with different parameters. Which command is executed depends on which parameters are supplied when the command is invoked. This can be used to overlay an existing command.

mount (cmdpathType) [] — The "mount" attribute is optional. It specifies where in the command hierarchy of built-in commands this command should be mounted. If no mount attribute is given, or if it is empty (""), the command is mounted on the top-level of the CLI hierarchy.

An example:

```
<cmd name="copy" mount="file">
  <info>Copy a file</info>
  <help>Copy a file from in the file system.</help>
  <callback>
    <exec>
      <osCommand>cp</osCommand>
      <options>
        <uid>confd</uid>
      </options>
    </exec>
  </callback>
  <params>
    <param>
      <type><file/></type>
```

```
      <info>&amp;lt;source file&amp;gt;</info>
    </param>
    <param>
      <type><file/></type>
      <info>&amp;lt;destination&amp;gt;</info>
    </param>
  </params>

  <cmd ...>
    ...
    <cmd ...>
      ...
    </cmd>
  </cmd>

  <cmd ...>
    ...
  </cmd>
</cmd>
```

`/clispec/$MODE/cmd/info (xs:string)`

The "info" element is a single text line describing the command.

An example:

```
<cmd name="start">
  <info>Start displaying the system log or trace a file</info>
  ...
```

and when we do the following in the CLI we get:

```
joe@xev> monitor st<TAB>
Possible completions:
  start - Start displaying the system log or trace a file
  stop  - Stop displaying the system log or trace a file
joe@xev> monitor st
```

`/clispec/$MODE/cmd/help (xs:string)`

The "help" element is a multi-line text string describing the command. This text is shown when we use the "help" command.

An example:

```
joe@xev> help monitor start
Help for command: monitor start
Start displaying the system log or trace a file in the background.
We can abort the logging using the "monitor stop" command.
joe@xev>
```

`/clispec/$MODE/cmd/timeout (xs:integer|infinity)`

The "timeout" element is a timeout for the command in seconds. Default is infinity.

`/clispec/$MODE/cmd/confirmText`

See /clispec/$MODE/modifications/confirmText

`/clispec/$MODE/cmd/callback`

The "callback" element specifies how the command is implemented, e.g. as a OS executable or a CAPI callback. It contains one of the elements "capi", "exec", "table" or "execStop".

Note: A command which has a callback defined may not have recursive sub-commands. Likewise, a command which has recursive sub-commands may not have a callback defined. A command without sub-commands must have a callback defined.

`/clispec/$MODE/cmd/callback/table`

The "table" element specifies that the command should display parts of the configuration in the form of a table.

An example:

```
<callback>
  <table>
    <root>/all:config/hosts/host</root>
    <item>
      <width>20</width>
      <header>NAME</header>
      <path>name</path>
      <align>lefg</align>
    </item>
    <item>
      <header>DOMAIN</header>
      <path>domain</path>
    </item>
    <item>
      <header>IP</header>
      <path>interfaces/interface/ip</path>
      <align>right</align>
    </item>
  </table>
</callback>
```

`/clispec/$MODE/cmd/callback/table/root` (xs:string)

Should be a path to a list element. All item paths in the table are relative to this path.

`/clispec/$MODE/cmd/callback/table/legend` (xs:string)

Should be a legend template to display before showing the table.

`/clispec/$MODE/cmd/callback/table/footer` (xs:string)

Should be a footer template to display after showing the table.

`/clispec/$MODE/cmd/callback/table/item`

Specifies a column in the table. It contains a "header" element and a "path" element, and optionally a "width" element.

`/clispec/$MODE/cmd/callback/table/item/header` (xs:string)

Header of this column in the table.

`/clispec/$MODE/cmd/callback/table/item/path` (xs:string)

Path to the element in this column.

`/clispec/$MODE/cmd/callback/table/item/width` (xs:integer)

The width in characters of this column.

`/clispec/$MODE/cmd/callback/table/item/align` (left|right|center)

The data alignment of this column.

`/clispec/$MODE/cmd/callback/capi`

The "capi" element specifies that the command is implemented using C-API using the same API as for actions. It contains one "cmdpoint" element.

An example:

```
<callback>
  <capi>
    <cmdpoint>adduser</cmdpoint>
  </capi>
</callback>
```

`/clispec/$MODE/cmd/callback/capi/cmdpoint` (xs:NCName)

The "cmdpoint" element specifies the name of the C-API action to be called. For this to work, a actionpoint must be registered with the ConfD daemon at startup.

`/clispec/$MODE/cmd/callback/exec`

The "exec" element specifies how the command is implemented using an executable or a shell script. It contains (in order) one "osCommand" element, zero or one "args" elements and zero or one "options" elements.

An example:

```
<callback>
  <exec>
    <osCommand>cp</osCommand>
    <options>
      <uid>confd</uid>
      <wd>/var/tmp</wd>
      ...
    </options>
  </exec>
</callback>
```

`/clispec/$MODE/cmd/callback/exec/osCommand` (xs:token)

The "osCommand" element specifies the path to the executable or shell script to be called. If the command is in the $PATH (as specified when we start the ConfD daemon) the path may just be the name of the command.

The command is invoked as if it had been executed by exec(3), i.e. not in a shell environment such as "/bin/sh -c ...".

`/clispec/$MODE/cmd/callback/exec/args` (argsType)

The "args" element specifies the arguments to use when executing the command specified by the "osCommand" element. argsType is a space-separated list of argument strings. The built-in variables are: "cwd", "user", "groups", "ip", "maapi", "uid", "gid", "tty", "ssh_connection", "opaque", "path", "cpath", "ipath" and "licounter". The variable "pipecmd_XYZ" can be used to determine whether a certain builtin pipe command has been run together with the command. Here XYZ is the name of the pipe command. An example of such a variable is "pipecmd_include". In addition the variables "spath" and "ispath" are available when a command is executed from a show path. For example:

```
<args>$(user)</args>
```

Will expand to the username.

`/clispec/$MODE/cmd/callback/exec/options`

The "options" element specifies how the command is be executed. It contains (in any order) zero or one "uid" elements, zero or one "gid" elements, zero or one "wd" elements, zero or one "batch" elements, zero or one of "interrupt" elements, and zero or one "ignoreExitValue" elements.

`/clispec/$MODE/cmd/callback/exec/options/uid` (idType) [confd]

The "uid" element specifies which user id to use when executing the command. Possible values are:

confd (default)	The command is run as the same user id as the ConfD daemon.
user	The command is run as the same user id as the user logged in to the CLI, i.e. we have to make sure that this user id exists as an actual user id on the device.
root	The command is run as root.
<*uid*> (the numerical user <*uid*>)	The command is run as the user id <uid>.
	Note: If uid is set to either "user", "root" or "<uid>" the the ConfD daemon must have been started as root (or setuid), or the cmdptywrapper must have setuid root permissions.

`/clispec/$MODE/cmd/callback/exec/options/gid` (idType) [confd]

The "gid" element specifies which group id to use when executing the command. Possible values are:

confd (default)	The command is run as the same group id as the ConfD daemon.
user	The command is run as the same group id as the user logged in to the CLI, i.e. we have to make

	sure that this group id exists as an actual group on the device.
root	The command is run as root.
<gid> (the numerical group *<gid>*)	The command is run as the group id <gid>. *Note:* If gid is set to either "user", "root" or "<gid>" the the ConfD daemon must have been started as root (or setuid), or the cmdptywrapper must have setuid root permissions.

`/clispec/$MODE/cmd/callback/exec/options/wd` (xs:token)

The "wd" element specifies which working directory to use when executing the command. If not given, the command is executed from the location of the CLI.

`/clispec/$MODE/cmd/callback/exec/options/pty` (xs:boolean)

The "pty" element specifies weather a pty should be allocated when executing the command. The default is to allocate a pty for operational and configure osCommands, but not for osCommands executing as a pipe command. This behavior can be overridden with this parameter.

`/clispec/$MODE/cmd/callback/exec/options/globalNoDuplicate` (xs:token)

The "globalNoDuplicate" element specifies that only one instance with the same name can be run at any one time in the system. The command can be started either from the CLI, the Web UI or through NETCONF.

`/clispec/$MODE/cmd/callback/exec/options/noInput` (xs:token)

The "noInput" element specifies that the command should not grab the input stream and consume freely from that. This option should be used if the command should not consume input characters. If not used then the command will eath all data from the input stream and cut-and-paste may not work as intended.

`/clispec/$MODE/cmd/callback/exec/options/batch`

The "batch" element makes it possible to specify that a command returns immediately but still runs in the background, optionally generating output on stdout. An example of such a command is the standard "monitor start" command, which prints additional data appended to a (log) file:

```
joe@io> monitor start /var/log/messages
joe@io>
log: Apr 10 11:59:32 earth ntpd[530]: kernel time sync enabled 2001
```

Ten seconds later...

```
log: Apr 12 01:59:02 earth sshd[26847]: error: PAM: auth error for cathy
joe@io> monitor stop /var/log/messages
joe@io>
```

The "batch" element contains (in order) one "group" element, an optional "prefix" element, and an optional "noDuplicate" element. The prefix defaults to the empty string.

An example from confd .cli implementing the monitor functionality:

```
<cmd name="start">
  ...
  <callback>
    <exec>
      <osCommand>tail</osCommand>
      <args>-f -n 0</args>
      <options>
        ...
        <batch>
          <group>monitor_file</group>
          <prefix>log:</prefix>
          <noDuplicate/>
        </batch>
      </options>
    </exec>
  </callback>
  ...
</cmd>
```

The batch group is used to kill the command as exemplified in the "execStop" element description below. "noDuplicate" indicates that a specific file is not allowed to be monitored by several commands in parallel.

`/clispec/$MODE/cmd/callback/exec/options/batch/group` (xs:NCName)

The "group" element attaches a group label to the command. The group label is used when defining a "stop" command whose job it is to kill the background command. Take a look at the monitor example above for better understanding.

The stop command is defined using a "execStop" element as described below.

`/clispec/$MODE/cmd/callback/exec/options/batch/prefix` (xs:NCName)

The "prefix" element specifies a string to prepend to all lines printed by the background command. In the monitor example above, "log:" is the chosen prefix.

`/clispec/$MODE/cmd/callback/exec/options/batch/noDuplicate`

The "noDuplicate" element specifies that only a single instance of this batch command, including the given/specified parameters, can run in the background.

`/clispec/$MODE/cmd/callback/exec/options/interrupt` (interruptType) [sigkill]

The "interrupt" element specifies what should happen when the user enters ctrl-c in the CLI. Possible values are:

sigkill (default)	The command is terminated by sending the sigkill signal.
sigint	The command is interrupted by the sigint signal.
sigterm	The command is interrupted by the sigterm signal.
ctrlc	The command is sent the ctrl-c character which is interpreted by the pty.

`/clispec/$MODE/cmd/callback/exec/options/ignoreExitValue`(xs:boolean) [false]

The "ignoreExitValue" element specifies if the CLI engine should ignore the fact that the command returns a non-zero value. Normally it signals an error on stdout if a non-zero value is returned.

/clispec/$MODE/cmd/callback/execStop

The "execStop" element specifies that a command defined by an "exec" element is to be killed.

Attributes:

batchGroup (xs:NCName) — The "batchGroup" attribute is mandatory. It specifies a background command to kill. It corresponds to a group label defined by another "exec" command using the "batch" element.

An example from confd.cli which kills a background monitor session:

```
<cmd name="stop">
  ...
  <callback>
    <execStop batchGroup="monitor_file"/>
  </callback>
  ...
</cmd>
```

/clispec/$MODE/cmd/params

The "params" element lists which parameters the CLI should prompt for. These parameters are then used as arguments to either the CAPI callback or the OS executable command (as specified by the "capi" element or the "exec" element, respectively). If an "args" element as well as a "params" element has been specified, all of them are used as arguments: first the "args" arguments and then the "params" values are passed to the CAPI callback or executable.

The "params" element contains (in order) zero or more "param" elements and zero or one "any" elements.

Attributes:

mode (list|choice) — This is an optional attribute. If it is "choice" then at least "min" and at most "max" params must be given by the user. If it is "list" then all non-optional parameters must be given the command in the order they appear in the list.

min (xs:nonNegativeInteger) — This optional attribute defines the minumun number of parameters from the body of the "params" element that the user must supply with the command. It is only applicable if the mode attribute has been set to "choice". The default value is "1".

max (xs:nonNegativeInteger | unlimited) — This optional attribute defines the maximum number of parameters from the body of the

"params" element that the user may supply with the command. It is only applicable if the mode attribute has been set to "choice". The default value is "1" unless multi is specified, in which case the default is "unlimited".

multi (xs:boolean) — This optional attribute controls if each parameters should be allowed to be entered more than once. If set to "true" then each parameter may occur multiple times. The default is "false".

An example from confd.cli which copies one file to another:

```
<params>
  <param>
    <type><file/></type>
    ...
  </param>
  <param>
    <type><file/></type>
    ...
   </param>
   ...
</params>
```

`/clispec/$MODE/cmd/params/param`

The "param" element defines the nature of a single parameter which the CLI should prompt for. It contains (in any order) zero or one "type" element, zero or one "info" element, zero or one "help" element, zero or one "optional" element, zero or one "name" element, zero or one "params" element, zero or one "auditLogHide" element, zero or one "prefix" element, zero or one "flag" element, zero or one "id" element, zero or one "hideGroup" element, and zero or one "simpleType" element and zero or one "completionId" element.

`/clispec/$MODE/cmd/params/param/type`

The "type" element is optional and defines the parameter type. It contains either a "enums", "enumerate", "void", "keypath", "file", "url_file", "simpleType", "xpath", "url_directory_file", "directory_file", "url_directory" or a "directory" element. If the "type" element is not present, the value entered by the user is passed unmodified to the callback.

`/clispec/$MODE/cmd/params/param/type/enums` (enumsType)

The "enums" element defines a list of allowed enum values for the parameter. enumsType is a space-separated list of string enums.

An example:

```
<enums>for bar baz</enums>
```

`/clispec/$MODE/cmd/params/param/type/enumerate`

The "enumerate" is used to define a set of values with info text. It can contain one of more of the element "elem".

`/clispec/$MODE/cmd/params/param/type/enumerate/enum`

The "enum" is used to define an enumeration value with help text. It must contain the element "name" and optionally an "info" element and a "hideGroup" element.

`/clispec/$MODE/cmd/params/param/type/enumerate/enum/name`(xs:token)

The "name" is used to define the name of an enumeration.

`/clispec/$MODE/cmd/params/param/type/enumerate/enum/info`(xs:string)

The "info" is used to define the info that is displayed during completion in the CLI. The element is optional.

`/clispec/$MODE/cmd/params/param/type/enumerate/enum/hideGroup`(xs:string)

The "hideGroup" element makes an enum value invisible and it cannot be used even if a user knows about its existence. The enum value will become visible when the hide group is 'unhidden' using the unhide command.

`/clispec/$MODE/cmd/params/param/type/void`

The "void" element is used to indicate that this parameter should not prompt for a value. It can only be used when the "name" element is used.

`/clispec/$MODE/cmd/params/param/type/keypath` (keypathType)

The "keypath" element specifies that the parameter must be a keypath pointing to a configuration value. Valid keypath values are: *new* or *exist*:

new The keypath is either an already existing configuration value or an instance value to be created.

exist The keypath must be an already existing configuration value.

`/clispec/$MODE/cmd/params/param/type/key` (path)

The "key" element specifies that the parameter is an instance identifier, either an existing instance or a new. If the list has multiple key elements then they will be entered with a space in between.

The path should point to a list element, not the actual key leaf. If the list has multiple keys then they user will be requested to enter all keys of an instance. The path may be either absolute or relative to the current submode path. Also variables referring to key elements in the current submode path may be used, where the closes key is named $(key-1-1), $(key-1-2) etc. Eg

```
/foo{key-2-1,key-2-2}/bar{key-1-1,key-1-2}/...
```

Attributes:

mode (keypathType) The "mode" attribute is mandatory. It specifies if the parameter refers to an existing (exist) instance or a new (new) instance.

`/clispec/$MODE/cmd/params/param/type/pattern` (patternType)

The "pattern" element specifies that the parameter must be a show command pattern. Valid pattern values are: *stats* or *config* or *all*:

stats The pattern is only related to "config false" nodes in the data model. Note that CLI modifications such as fullShowPath, incompleteShowPath etc are applied to this pattern.

config The pattern is only related to "config true" elements in the data model.

all The pattern spans over all visible nodes in the data model.

`/clispec/$MODE/cmd/params/param/type/file`

The "file" element specifies that the parameter is a file on disk. The CLI automatically enables tab completion to help the user to choose the correct file.

Attributes:

wd (xs:token) The "wd" attribute is optional. It specifies a working directory to be used as the root for the tab completion algorithm. If no "wd" attribute is specified, the working directory is as defined for the "/clispec/$MODE/cmd/callback/exec/options/wd" element.

An example:

```
<file wd="/var/log/"/>
```

`/clispec/$MODE/cmd/params/param/type/url_file`

The "url_file" element specifies that the parameter is a file on disk or an URL. The CLI automatically enables tab completion to help the user to choose the correct file.

Attributes:

wd (xs:token) The "wd" attribute is optional. It specifies a working directory to be used as the root for the tab completion algorithm. If no "wd" attribute is specified, the working directory is as defined for the "/clispec/$MODE/cmd/callback/exec/options/wd" element.

An example:

```
<file wd="/var/log/"/>
```

`/clispec/$MODE/cmd/params/param/type/simpleType`

The "simpleType" element specifies that the parameter should conform to some specific simpleType specified in a namespace. It can contain zero or one "info" element

Attributes:

namespace (xs:string) The "namespace" attribute is required. It specifies in which namespace the type is found. It can be either the namespace URI or the namespace prefix.

name (xs:string) The "name" attribute is required. It specifies the name of the type in the given namespace.

disallowValue (xs:string) The "disallowValue" attribute is optional. It specifies a regular expression of unaccepted values.

An example:

```
<simpleType namespace="<ulink url='http://tail-f.com/ns/confd/1.0'>http://tail-f.com/
           name="inetAddressIP"/>
```

/clispec/$MODE/cmd/params/param/type/simpleType/info

The "info" element contains a single line describing the simpleType that will appear during auto-completion. Note that this will override any other info texts provided by this type.

/clispec/$MODE/cmd/params/param/type/xpath

The "xpath" element specifies that the parameter should conform to one of the values returned by the xpath expression given as attribute.

Attributes:

expr (xs:string) The "expr" attribute is required. It specifies an xpath expression that returns a set of valid values for this parameter. The expression may contain variables defined in the dict.

ctx (xs:string) The "ctx" attribute is optional. It specifies the context for the evaluation of the xpath expression. The path may contain variables defined in the dict.

lax (xs:boolean) The "lax" attribute is optional. It specifies if the given value should be checked against the values given by the xpath expression. The default is "true" which means that tab completion will present the values given by the xpath expression but the parser will accept any value. This makes parsing a bit faster. When lax is set to "false" a syntax error will be generated if an unexisting value is given as parameter.

An example:

```
<params>
  <param>
    <name>host</name>
    <type>
      <xpath expr="/simpleObjects/hosts/host/name" lax="false" />
    </type>
    <flag>--host</flag>
    <id>hostname</id>
  </param>
  <param>
    <name>server</name>
    <type>
      <xpath expr="servers/server/name"
             ctx="/simpleObjects/hosts/host{$(hostname)}"/>
    </type>
    <flag>--server</flag>
  </param>
```

```
</params>
```

/clispec/$MODE/cmd/params/param/type/directory

The "directory" element specifies that the parameter is a directory on disk. The CLI automatically enables tab completion to help the user choose the correct directory.

Attributes:

wd (xs:token) The "wd" attribute is optional. It specifies a working directory to be used as the root for the tab completion algorithm. If no "wd" attribute is specified, the working directory is as defined for the "wd" element.

An example:

```
<directory wd="/var/log/"/>
```

/clispec/$MODE/cmd/params/param/type/url_directory

The "url_directory" element specifies that the parameter is a directory on disk or an URL. The CLI automatically enables tab completion to help the user choose the correct directory.

Attributes:

wd (xs:token) The "wd" attribute is optional. It specifies a working directory to be used as the root for the tab completion algorithm. If no "wd" attribute is specified, the working directory is as defined for the "wd" element.

An example:

```
<directory wd="/var/log/"/>
```

/clispec/$MODE/cmd/params/param/type/directory_file

The "directory_file" element specifies that the parameter is a directory or a file on disk. The CLI automatically enables tab completion to help the user choose the correct directory or file.

An example:

```
<directory_file/>
```

/clispec/$MODE/cmd/params/param/type/url_directory_file

The "url_directory_file" element specifies that the parameter is a directory or a file on disk or an URL. The CLI automatically enables tab completion to help the user choose the correct directory or file.

An example:

```
<directory_file/>
```

/clispec/$MODE/cmd/params/param/info (xs:string)

The "info" element is a single text line describing the parameter.

An example:

```
<cmd name="id" mount="">
  <info>Find uid and groups of a user</info>
  <help>Find uid and groups of a user, using the id program</help>
  <callback>
    <exec>
      <osCommand>id</osCommand>
    </exec>
  </callback>
  <params>
    <param>
      <info>User name</info>
      <help>User name</help>
    </param>
  </params>
</cmd>
```

and when we do the following in the CLI we get:

```
joe@x15> id <TAB>
User name
joe@x15> id snmp
uid=108(snmp) gid=65534(nogroup) groups=65534(nogroup)
[ok][2006-08-30 14:51:28]
```

Note: This description is *only* shown if the "type" element is left out.

/clispec/$MODE/cmd/params/param/help (xs:string)

The "help" element is a multi-line text string describing the parameter. This text is shown when we use the '?' character.

/clispec/$MODE/cmd/params/param/hideGroup (xs:string)

The "hideGroup" element makes a CLI parameter invisible and it cannot be used even if a user knows about its existence. The parameter will become visible when the hide group is 'unhidden' using the unhide command.

This mechanism correspond to the 'tailf:hidden' statement in a YANG module.

/clispec/$MODE/cmd/params/param/name (xs:token)

The "name" element is a token which has to be entered by the user before entering the actual parameter value. It is used to get named parameters.

An example:

```
<cmd name="copy" mount="file">
  <info>Copy a file</info>
  <help>Copy a file from one location to another in the file system</help>
  <callback>
    <exec>
```

```
      <osCommand>cp</osCommand>
      <options>
        <uid>user</uid>
      </options>
    </exec>
  </callback>
  <params>
    <param>
      <type><file/></type>
      <info>&amp;lt;source file&amp;gt;</info>
      <help>source file</help>
      <name>from</name>
    </param>
    <param>
      <type><file/></type>
      <info>&amp;lt;destination file&amp;gt;></info>
      <help>destination file</help>
      <name>to</name>
    </param>
  </params>
</cmd>
```

The result is that the user has to enter

```
file copy from /tmp/orig to /tmp/copy
```

/clispec/$MODE/cmd/params/param/prefix (xs:string)

The "prefix" element is a string that is prepended to the argument before calling the osCommand. This can be used to add Unix style command flags in front of the supplied parameters.

An example:

```
<cmd name="ssh">
  <info>Open a secure shell on another host</info>
  <help>Open a secure shell on another host</help>
  <callback>
    <exec>
      <osCommand>ssh</osCommand>
      <options>
        <uid>user</uid>
        <interrupt>ctrlc</interrupt>
      </options>
    </exec>
  </callback>
  <params>
    <param>
      <info>&amp;lt;login&amp;gt;</info>
      <help>Users login name on host</help>
      <name>user</name>
      <prefix>--login=</prefix>
    </param>
    <param>
      <info>&amp;lt;host&amp;gt;</info>
      <help>host name or IP</help>
      <name>host</name>
    </param>
```

```
  </params>
</cmd>
```

The user would enter for example

```
ssh user joe host router.intranet.net
```

and the resulting call to the ssh executable would become

```
ssh --login=joe router.intranet.net
```

/clispec/$MODE/cmd/params/param/flag (xs:string)

The "flag" element is a string that is prepended to the argument before calling the osCommand. In contrast to the prefix element it will not be appended to the current parameter, but instead appear as a separate argument, ie instead of adding a unix style flag as "--foo=" (prefix) you add arguments in the style of "-f <param>" where -f is one arg and <param> is another. Both <prefix> and <flag> can be used at the same time.

An example:

```
<cmd name="ssh">
  <info>Open a secure shell on another host</info>
  <help>Open a secure shell on another host</help>
  <callback>
    <exec>
      <osCommand>ssh</osCommand>
      <options>
        <uid>user</uid>
        <interrupt>ctrlc</interrupt>
      </options>
    </exec>
  </callback>
  <params>
    <param>
      <info>&amp;lt;login&amp;gt;</info>
      <help>Users login name on host</help>
      <name>user</name>
      <flag>-l</flag>
    </param>
    <param>
      <info>&amp;lt;host&amp;gt;</info>
      <help>host name or IP</help>
      <name>host</name>
    </param>
  </params>
</cmd>
```

The user would enter for example

```
ssh user joe host router.intranet.net
```

and the resulting call to the ssh executable would become

```
ssh -l joe router.intranet.net
```

`/clispec/$MODE/cmd/params/param/id (xs:string)`

The "id" is used for identifying the value of the parameter and can be used as a variable in the value of a key parameter.

An example:

```
<cmd name="test">
  <info/>
  <help/>
  <callback>
    <exec>
      <osCommand>/bin/echo</osCommand>
    </exec>
  </callback>
  <params>
    <param>
      <name>host</name>
      <id>h</id>
      <type><key mode="exist">/host</key></type>
    </param>
    <param>
      <name>interface</name>
      <type><key mode="exist">/host{$(h)}/interface</key></type>
    </param>
  </params>
</cmd>
```

There are also three builtin variables: user, uid and gid. The id and the builtin variables can be used in when specifying the path value of a key parameter, and also when specifying the wd attribute of the file, url_file, directory, and url_directory.

`/clispec/$MODE/cmd/params/param/callback/capi`

Specifies that the parameter completion should be calculated through a callback function. It contains exactly one "completionpoint" element.

`/clispec/$MODE/cmd/params/param/callback/capi/completionpoint` (xs:string)

Specifies the callpoint name of the completion function.

`/clispec/$MODE/cmd/params/param/auditLogHide`

The "auditLogHide" element specifies that the parameter should be obfuscated in the audit log. This is suitable when clear text passwords are passed as command parameters.

`/clispec/$MODE/cmd/params/param/optional`

The "optional" element specifies that the parameter is optional and not required. It contains zero or one "default" element. It cannot be used inside a params of type "choice".

`/clispec/$MODE/cmd/params/param/optional/default`

The "default" element makes it possible to specify a default value, should the parameter be left out.

An example:

```
<optional>
    <default>42</default>
</optional>
```

`/clispec/$MODE/cmd/params/param/completionId` xs:string

The "completionId" element makes it possible to identify a specific parameter whenever it is referred to from a completion callback, i.e. a completion callback takes an optional completion ID parameter as input. Read more about completion callbacks in the confd_lib_dp(3) manual page and in the "The CLI agent" User Guide chapter.

`/clispec/$MODE/cmd/params/any`

The "any" element specifies that any number of parameters are allowed. It contains (in any order) one "info" element and one "help" element.

`/clispec/$MODE/cmd/params/any/info (xs:string)`

The "info" element is a single text line describing the parameter(s) expected.

An example:

```
<cmd name="evaluate" mount="">
  <info>Evaluate an arithmetic expression</info>
  <help>Evaluate an arithmetic expression, using the expr program</help>
  <callback>
    <exec>
      <osCommand>expr</osCommand>
    </exec>
  </callback>
  <params>
    <any>
      <info>Arithmetic expression</info>
      <help>Arithmetic expression</help>
    </any>
  </params>
</cmd>
```

and when we do the following in the CLI we get:

```
joe@xev> eva<TAB>
joe@xev> evaluate <TAB>
Arithmetic expression
joe@xev> evaluate 2 + 5
7
[ok][2006-08-30 14:47:17]
```

`/clispec/$MODE/cmd/params/any/help (xs:string)`

The "help" element is a multi-line text string describing these anonymous parameters. This text is shown we use the '?' character.

`/clispec/$MODE/cmd/options`

The "options" element specifies under what circumstances the CLI command should execute. It contains (in any order) zero or one "hidden" element, zero or one "hideGroup"

element, zero or one "denyRunAccess" element, zero or one "notInterruptible" element, zero or one "pipeFlags" element, zero or one "negPipeFlags" element, zero or one of "submodeCommand" and "topModeCommand", zero or one of "displayWhen" element, and zero or one "paginate" element.

`/clispec/$MODE/cmd/options/hidden`

The "hidden" element makes a CLI command invisible even though it can be evaluated if we know about its existence. This comes handy for commands which are used for debugging or are in pre-release state.

`/clispec/$MODE/cmd/options/hideGroup (xs:string)`

The "hideGroup" element makes a CLI command invisible and it cannot be used even if a user knows about its existence. The command will become visible when the hide group is 'unhidden' using the unhide command.

This mechanism correspond to the 'tailf:hidden' statement in a YANG module.

`/clispec/operationalMode/cmd/options/denyRunAccess`

The "denyRunAccess" element is used to restrict the possibility to run an operational mode command from configure mode.

Comment: The built-in "run" command is used to execute operational mode commands from configure mode.

`/clispec/$MODE/cmd/options/displayWhen`

The "displayWhen" element can be used to add a displayWhen XPath condition to a command.

Attributes:

exrp (xpath expression)	The "expr" attribute is mandatory. It specifies an xpath expression. If the expression evaluates to true then the command is available, otherwise not.
ctx (path)	The "ctx" attribute is optional. If not specified the current editpath/mode-path is used as context node for the xpath evaluation. Note that the xpath expression will automatically evaluate to false if a display when expression is used for a top-level command and no ctx is specified. The path may contain variables defined in the dict.

`/clispec/$MODE/cmd/options/notInterruptible`

The "notInterruptible" element disables <ctrl-c> and the execution of the CLI command can thus not be interrupted.

`/clispec/$MODE/cmd/options/submodeCommand`

The "submodeCommand" element makes a CLI command visible in all submodes.

`/clispec/$MODE/cmd/options/topModeCommand`

The "topModeCommand" element prevents a command mounted under a top mode command from being visible in all submodes.

`/clispec/$MODE/cmd/options/pipeFlags`

The "pipeFlags" element is used to signal that certain pipe commands should be made available if this command is entered.

`/clispec/$MODE/cmd/options/negPipeFlags`

The "negPipeFlags" element is used to signal that certain pipe commands should not be made available if this command is entered, ie it is used to block out specific pipe commands.

`/clispec/$MODE/cmd/options/paginate`

The "paginate" element enables a filter for paging through CLI command output text one screen at a time.

SEE ALSO

The ConfD User Guide confdc(1) - Confdc compiler confd_lib_dp(3) - callback library for connecting to ConfD confd.cli - The standard ConfD CLI commands. clispec.xsd - A W3C XML schema (`http://tail-f.com/ns/clispec/1.0`) describing a clispec.

Name

confd.conf — ConfD daemon configuration file format

DESCRIPTION

Whenever we start (or reload) the ConfD daemon it reads its configuration from `/etc/confd/confd.conf` or from the file specified with the `-c` option, as described in confd(1).

`confd.conf` is an XML configuration file formally defined by a W3C XML Schema, as referred to in the SEE ALSO section. This schema file is included in the distribution. The ConfD distribution also includes a commented confd.conf.example file.

Tip

In the ConfD distribution there is an Emacs mode suitable for confd.conf editing.

A short example: A ConfD configuration file which specifies where to find compiled YANG files etc, which facility to use for syslog, that the developer log should be disabled and that the audit log should be enabled. Finally, it also disables clear text NETCONF support:

```
<?xml version="1.0" encoding="UTF-8"?>
<confdConfig xmlns="http://tail-f.com/ns/confd_cfg/1.0">
  <loadPath>
    <dir>/etc/confd</dir>
  </loadPath>

  <stateDir>/var/confd/state</stateDir>

  <cdb>
    <enabled>true</enabled>
    <dbDir>/var/confd/cdb</dbDir>
  </cdb>

  <aaa>
    <sshServerKeyDir>/etc/confd/ssh</sshServerKeyDir>
  </aaa>

  <datastores>
    <startup>
      <enabled>false</enabled>
    </startup>
    <candidate>
      <filename>/var/confd/candidate/candidate.db</filename>
    </candidate>
  </datastores>

  <logs>
    <syslogConfig>
      <facility>daemon</facility>
    </syslogConfig>
    <developerLog>
      <enabled>false</enabled>
    </developerLog>
    <auditLog>
      <enabled>true</enabled>
    </auditLog>
  </logs>
```

```
  <netconf>
    <transport>
      <tcp>
        <enabled>false</enabled>
      </tcp>
    </transport>
  </netconf>

  <webui>
    <transport>
      <tcp>
        <enabled>false</enabled>
        <ip>0.0.0.0</ip>
        <port>8008</port>
      </tcp>
    </transport>
  </webui>
</confdConfig>
```

Many configuration parameters get their default values as defined in the schema. Filename parameters have no default values.

CONFIGURATION PARAMETERS

This section lists all available configuration parameters and their type (within parenthesis) and default values (within square brackets). Parameters are written using a path notation to make it easier to see how they relate to each other:

`/confdConfig`
The top-level element.

`/confdConfig/confdIpcAddress/ip`
ConfD listens by default on 127.0.0.1:4565 for incoming TCP connections from CDB, MAAPI, the CLI, the external database API, as well as commands from the confd script (such as "confd --reload").

This value and port (below) can be changed. If they are changed all clients using MAAPI, CDB et.c. must be re-compiled to handle this. See the ConfD IPC section in the Advanced Topics chapter in the User Guide on how to do this.

Note that there are severe security implications involved if ConfD is instructed to bind(2) to anything but localhost. Read more about this in the ConfD IPC section in the Advanced Topics chapter in the User Guide. Use the IP 0.0.0.0 if you want ConfD to to listen(2) on all IPv4 addresses.

`/confdConfig/confdIpcAddress/port`
The port number where ConfD listens for incoming connections from CDB, the CLI and the external database API.

`/confdConfig/confdIpcExtraListenIp` (confd:inetAddressIP)
A list of additional IPs to which we wish to bind the ConfD IPC listener. This is useful if we don't want to use the wildcard 0.0.0.0 address in order to never expose the ConfD IPC to certain interfaces.

`/confdConfig/confdExternalIpc/enabled` (boolean) [false]
Enables a user-provided IPC mechanism. ConfD can be set up to use a different protocol than TCP for the IPC connections, see the ConfD IPC section in the Advanced Topics chapter in the User Guide for the details.

`/confdConfig/confdExternalIpc/address` (xs:string)
The address where ConfD should listen for incoming connections from CDB, MAAPI, etc, when the user-provided IPC mechanism is enabled. See the ConfD IPC section in the Advanced Topics chapter in the User Guide for further details.

`/confdConfig/confdIpcListenBacklog` (xs:int) [25]
The maximum length to which the queue of pending connections for the IPC sockets may grow (see the OS manual page for listen(2)). If a very large number of applications connect to ConfD more or less simultaneously at startup, this value may need to be raised to avoid connection failures.

Note

The OS may restrict the length to a lower value, e.g. on Linux it is silently truncated to the value in `/proc/sys/net/core/somaxconn` - i.e. this value may also need to be raised.

`/confdConfig/confdIpcAccessCheck/enabled` (boolean) [false]
Enables access check for incoming connections to the IPC listener sockets. The access check requires that connecting clients prove possession of a shared secret. See the section Restricting access to the IPC port in the Advanced Topics chapter in the User Guide for the details.

`/confdConfig/confdIpcAccessCheck/filename` (xs:string)
The full path to a file containing the shared secret for the IPC access check. The file should be protected via OS file permissions, such that it can only be read by the ConfD daemon and client processes that are allowed to connect to the IPC listener sockets. See the section Restricting access to the IPC port in the Advanced Topics chapter in the User Guide for further details.

`/confdConfig/runtimeReconfiguration (config-file|namespace) [config-file]`
Controls whether ConfD should find run-time modifiable configuration parameters in the configuration file (config-file setting) or whether ConfD should them from a namespace with the data stored in CDB. See further the Advanced Topics chapter in the Users Guide as well as the confdconf/dyncfg example in the example collection.

`/confdConfig/ignoreBindErrors/enabled` (boolean) [false]
enabled is either "true" or "false". By default (false) Confd will refuse to start if any of its nortbound agents fails to bind their respective ports. When enabled, this parameter forces Confd to ignore that fatal error situation and instead it just issues a warning and disables the failing agent. The agent may be enabled by dynamically re-configuring the failing agent to use another port and restart Confd.

`/confdConfig/loadPath/dir`
The loadPath element contains any number of dir elements. Each dir element points to a directory path on disk which is searched for compiled YANG files (.fxs files), and compiled clispec files (.ccl file) during daemon startup.

`/confdConfig/stateDir`
This is where ConfD writes persistent state data. Currently the only state files are 'running.invalid' which exists only if the running database status is invalid, which it will be if one of the database implementation fails during the two-phase commit protocol, and 'global.data' which is used to store some data that needs to be retained across reboots.

`/confdConfig/commitRetryTimeout`(xs:duration|infinity) [PT0S]
Commit timeout in the ConfD backplane. This timeout controls for how long the commit operation will attempt to complete the operation when some other entity is locking the database, e.g. some other commit is in progress or some managed object is locking the database.

`/confdConfig/maxValidationErrors`(xs:unsignedInt|unbounded) [1]
Controls how many validation errors are collected and presented to the user at a time when the user performs a validate or commit operation. Note that syntactical errors are detected and reported when the data is entered, and thus not covered by this parameter.

`/confdConfig/hideGroup - container element`
Hide groups that can be unhidden must be listed here. Multiple hideGroup entries are allowed in the confd.conf file.

If a hide groups does not have a hideGroup entry, then it cannot be unhidden using the CLI 'unhide' command. However, it is possible to add a hideGroup entry to the confd.conf file and then use confd --reload to make it available in the CLI. This may be useful to enable for example a diagnostics hide groups that you do not event want accessible using a password.

`/confdConfig/hideGroup/name (string)`
Name of hide group. This name should correspond to a hide group name in some data model.

`/confdConfig/hideGroup/password (string) []`
A password can optionally be specified for a hide group. If no password or callback is given then the hide group can be unhidden without giving a password.

If a password is specified then the hide group cannot be enabled unless the password is entered.

To completely disable a hide group, ie make it impossible to unhide it, remove the entire hideGroup container for that hide group.

`/confdConfig/hideGroup/callback (string) []`
A callback can optionally be specified for a hide group. If no callback or password is given then the hide group can be unhidden without giving a password.

If a callback is specified then the hide group cannot be enabled unless a password is entered and the callback successfully verifies the password. Using a callback it is possible to have short lived unhide passwords and per-user unhide passwords.

The callback must be registered as a `command()` callback with `confd_register_action_cbs()`, see confd_lib_dp(3). The *`path`* argument to the callback is always `"hidegroup"`, while *`argv[0]`* is the name of the hide group, *`argv[1]`* is the name of the user issuing the unhide command, and *`argv[2]`* is the given password. The callback should return CONFD_OK to allow the unhiding, otherwise CONFD_ERR.

`/confdConfig/cdb/enabled` (boolean) [false]
enabled is either "true" or "false". If "false", CDB is disabled.

`/confdConfig/cdb/persistent` (boolean) [true]
If persistent is set to false CDB will operate in RAM-only mode. This is only applicable for permanent slave nodes, i.e. slaves that are unable to become master, in a HA cluster.

`/confdConfig/cdb/journalCompaction` (automatic|manual) [automatic]
Controls the way the CDB configuration store does its journal compaction. Never set to anything but the default 'automatic' unless there is an external mechanism which controls the compaction using the cdb_initiate_journal_compaction() API call.

`/confdConfig/cdb/dbDir` (string)
dbDir is the directory on disk which CDB use for its storage and any temporary files being used. It is also the directory where CDB searches for initialization files.

`/confdConfig/cdb/initPath`
The initPath can contain any number of `<dir>` items, which should be directories. When CDB first starts it will first look in these directories for initialization files. The directories will be searched in the order they are listed, lastly the `dbDir` is searched.

`/confdConfig/cdb/clientTimeout` (xs:duration|infinity) [infinity]
clientTimeout specifies how long CDB should wait while a client performs a certain action, before considering that client unresponsive. When set to infinity, CDB will never timeout waiting for a response from a client. A client which doesn't respond will have its socket closed. The timeout is applied to clients in the following situations:

- When a reader client calls `cdb_start_session()` it must end it with `cdb_end_session()` within the timeout period.
- When a subscription client receives a subscription notification, it must respond with a call to `cdb_sync_subscription_socket()` within the timeout period.

`/confdConfig/cdb/subscriptionReplay/enabled` (xs:boolean) [false]
By setting `subscriptionReplay/enabled` to true it is possible to use the `cdb_replay_subscriptions()` to "replay" the previously committed transaction to CDB subscribers. This means that CDB subscribers that miss one subscription notification can have it triggered again. CDB will save the previous transaction in a separate file in the dbDir.

`/confdConfig/cdb/replication` (async|sync) [sync]
When CDB replication is enabled (which it is when high-availability mode is enabled, see /confdConfig/ha) the CDB configuration stores can be replicated either asynchronously or synchronously. With asynchronous replication, a transaction updating the configuration is allowed to complete as soon as the updates have been sent to the connected slaves. With the default synchronous replication, the transaction is suspended until the updates have been completely propagated to the slaves, and the subscribers on the slaves (if any) have acknowledged their subscription notifications (see confd_lib_cdb(3)).

`/confdConfig/cdb/operational - container element`
Operational data can either be implemented by external callbacks, or stored in CDB (or a combination of both). The operational datastore is used when data is to be stored in CDB.

`/confdConfig/cdb/operational/enabled` (xs:boolean) [false]
Whether to enable storage of operational data in CDB.

`/confdConfig/cdb/operational/dbDir` (string)
By default CDB operational uses the same directory for its storage. Use this setting to make CDB operational use a separate directory.

`/confdConfig/cdb/operational/persistent` (confspec|always|never) [confspec]
By default the decision on how operational data in CDB is stored (persistent or volatile) is decided for each element in the YANG data model, via the `tailf:persistent` substatement to `tailf:cdb-oper`, see tailf_yang_extensions(5) It is possible to override this by using this setting in confd.conf. If "never", CDB will only keep the operational datastore in RAM. And if set to "always" all CDB stored operational data will be persistently backed to a file.

`/confdConfig/cdb/operational/replication` (always|never|persistent) [persistent]
When CDB replication is enabled (which it is when high-availability mode is enabled, see /confdConfig/ha) the CDB operational store can optionally be replicated too. When set to "persistent", only persistent operational data is replicated. When set to "never", CDB operational is never replicated. Using "always" means that both persistent and non-persistent data is replicated. Note however that non-persistent data is only replicated to connected slaves at the time of writing. I.e. when a new slave connects to the master, only the persistent data is synchronized from the master, even if "always" is used.

`/confdConfig/cdb/operational/replicationMode` (async|sync) [async]
When CDB replication is enabled (which it is when high-availability mode is enabled, see /confdConfig/ha) the replication of the CDB operational store (according to `/confdConfig/cdb/operational/replication`) can be done either asynchronously or synchronously. With the default asynchronous replication, an API call writing operational data will return as soon as the updates have been sent to the connected slaves. With synchronous replication, the API call will block until the updates have been completely propagated to the slaves.

`/confdConfig/ha/enabled` (xs:boolean) [false]
Enables the high-availability mode.

`/confdConfig/ha/ip` (confd:inetAddressIP) [0.0.0.0]
Defines which IP address ConfD should use for incoming requests from other HA nodes.

`/confdConfig/ha/port` (confd:inetPortNumber) [4569]
Defines which port number confd should use for incoming requests from other HA nodes.

`/confdConfig/ha/externalIpc/enabled` (boolean) [false]
Enables a user-provided IPC mechanism for the communication between HA nodes. See the ConfD IPC section in the Advanced Topics chapter in the User Guide for further details.

`/confdConfig/ha/externalIpc/address` (xs:string)
The address ConfD should use for incoming requests from other HA nodes when the user-provided IPC mechanism is enabled. See the ConfD IPC section in the Advanced Topics chapter in the User Guide for further details.

`/confdConfig/ha/tickTimeout` (xs:duration) [PT20S]
Defines the timeout between keepalive ticks sent between HA nodes. The special value "PT0" means that no keepalive ticks will ever be sent.

`/confdConfig/encryptedStrings - container element`
encryptedStrings defines keys used to encrypt strings adhering to the types tailf:des3-cbc-encrypted-string and tailf:aes-cfb-128-encrypted-string as defined in the tailf-common YANG module, see the confd_types(3) manual page.

`/confdConfig/encryptedStrings/DES3CBC/key1` (hex8Value), `/confdConfig/encryptedStrings/DES3CBC/key2` (hex8Value), `/confdConfig/encryptedStrings/DES3CBC/key3` (hex8Value), `/confdConfig/encryptedStrings/DES3CBC/initVector` (hex8Value)
In the DES3CBC case three 64 bits (8 bytes) keys and a 64 bits (8 bytes) initial vector is used to encrypt the string.

`/confdConfig/encryptedStrings/AESCFB128/key` (hex16Value), `/confdConfig/encryptedStrings/AESCFB128/initVector` (hex16Value)
In the AESCFB128 case one 128 bits (16 bytes) key and a 128 bits (16 bytes) initial vector is used to encrypt the string.

`/confdConfig/cryptHash - container element`
cryptHash specifies how cleartext values should be hashed for leafs of the types ianach:crypt-hash, tailf:sha-256-digest-string, and tailf:sha-512-digest-string - see the confd_types(3) manual page.

`/confdConfig/cryptHash/algorithm` (md5|sha-256|sha-512) [md5]
algorithm can be set to one of the values 'md5', 'sha-256', or 'sha-512', to choose the corresponding hash algorithm for hashing of cleartext values for the ianach:crypt-hash type.

`/confdConfig/cryptHash/rounds` (xs:unsignedInt) [5000]
For the 'sha-256' and 'sha-512' algorithms for the ianach:crypt-hash type, and for the tailf:sha-256-digest-string and tailf:sha-512-digest-string types, 'rounds' specifies how many times the hashing loop

should be executed. If a value other than the default 5000 is specified, the hashed format will have 'rounds=N$', where N is the specified value, prepended to the salt. This parameter is ignored for the 'md5' algorithm for ianach:crypt-hash.

`/confdConfig/logs - container element`
This section defines settings which affect the logging done by ConfD.

`/confdConfig/logs/syslogConfig`
Shared settings for how to log to syslog. Logs (see below) can be configured to log to file and/or syslog. If a log is configured to log to syslog, the settings under /confdConfig/logs/syslogConfig are used used.

`/confdConfig/logs/syslogConfig/version` (bsd|1) [bsd]
version is either "bsd" (traditional syslog) or "1" (new IETF syslog format: RFC 5424). "1" implies that /confdConfig/logs/syslogConfig/udp/enabled must be set to true.

`/confdConfig/logs/syslogConfig/facility` (daemon|authpriv|local0|...|local7|xs:unsignedInt) [daemon]
facility is either "daemon", "authpriv", "local0", ..., "local7" or an unsigned integer. This facility setting is the default facility and applies if no explicit facility is set for a log. It's also possible to set individual facilities in the different logs below. Furthermore with the syslogServers container described below it is possible to set default facility on a per server basis. If facility is explicitly set for a log type, that item is used.

`/confdConfig/logs/syslogConfig/udp` container
Is a container for UDP syslog. This container can only contain the configuration for a single UDP syslog server. If we need more than one syslog server we must use the /confdConfig/logs/syslogConfig/syslogServers container instead. If the udp container is used, the syslogServers container is ignored.

`/confdConfig/logs/syslogConfig/udp/enabled` (boolean) [false]
enabled is either "true" or "false". If "false", messages will be sent to the local syslog daemon.

`/confdConfig/logs/syslogConfig/udp/host` (confd:inetAddress)
host is either a domain name or an IPv4/IPv6 network address. UDP syslog messages are sent to this host.

`/confdConfig/logs/syslogConfig/udp/port` (confd:inetPortNumber) [514]
port is a valid port number to be used in combination with /confdConfig/logs/syslogConfig/udp/host.

`/confdConfig/logs/syslogConfig/syslogServers` - container
We can have an arbitrary long list of syslog servers defined. As mentioned above, the use of the syslogServers container is an complementary alternative to the udp container.

`/confdConfig/logs/syslogConfig/syslogServers/server/host` (confd:inetAddress)
host is either a domain name or an IPv4/IPv6 network address. UDP syslog messages are sent to this host.

`/confdConfig/logs/syslogConfig/syslogServers/server/port` (confd:inetPortNumber) [514]
port is a valid port number to be used in combination with this syslog server

`/confdConfig/logs/syslogConfig/syslogServers/server/version` (bsd|1) [bsd]
Version of syslog messages for this syslog server.

`/confdConfig/logs/syslogConfig/syslogServers/server/facility` (daemon|local0|...|local7|xs:unsignedInt) [daemon]
Facility of syslog messages for this syslog server.

`/confdConfig/logs/syslogConfig/syslogServers/server/enabled` (true|false) [true]
Is this syslog server enabled.

`/confdConfig/logs/confdLog - container element`
confdLog is ConfD's daemon log. Check this log for startup problems of the ConfD daemon itself. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/confdLog/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the log is enabled.

`/confdConfig/logs/confdLog/file/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", file logging is enabled.

`/confdConfig/logs/confdLog/file/name` (xs:string)
name is the full path to the actual log file.

`/confdConfig/logs/confdLog/syslog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", syslog messages are sent.

`/confdConfig/logs/confdLog/syslog/facility` (daemon|authpriv|local0|...|local7|xs:unsignedInt)*[]*
facility is either "daemon", "authpriv", "local0", ..., "local7" or an unsigned integer. This optional value overrides the /confdConfig/logs/syslogConfig/facility for this particular log

`/confdConfig/logs/developerLog - container element`
developerLog is a debug log for troubleshooting user-written C code, AAA rules etc. Enable and check this log for problems with validation code etc. This log is enabled by default. In all other regards it can be configured as confdLog. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/developerLogLevel` (error|info|trace) [info]
Controls which level of developer messages are printed in the developer log.

`/confdConfig/logs/auditLog - container element`
auditLog is an audit log recording successful and failed logins to the ConfD backplane. This log is enabled by default. In all other regards it can be configured as /confdConfig/logs/confdLog. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/auditLogCommit` (xs:boolean) [false]
Controls whether the audit log should include messages about the resulting configuration changes for each commit to the running data store. If set to "true", the audit log will include entries of the form:

```
commit thandle <N> begin [confirmed [extended]]
commit thandle <N> <path> created
commit thandle <N> <path> deleted
commit thandle <N> <path> moved first
commit thandle <N> <path> moved after {<keys>}
commit thandle <N> <path> set to "<value>"
commit thandle <N> <path> default set (<value>)
commit thandle <N> <path> attribute "<name>" set to "<value>"
commit thandle <N> <path> attribute "<name>" deleted
commit thandle <N> end
commit confirmed completed
commit confirmed canceled
```

The "commit thandle <N> begin" entry indicates the start of a commit for the transaction with handle N. This is followed by a number of "commit thandle <N> <path> ..." entries detailing the changes,

and finally a "commit thandle <N> end" entry. If "begin" is followed by "confirmed", it means that the changes are part of a confirmed commit that will not be permanent until a "commit confirmed completed" entry is logged - if "commit confirmed canceled" is logged instead, the changes have been reverted. If "begin confirmed" is followed by "extended", it means that the changes are part of a confirmed commit that extends a confirmed commit that is already in progress.

`/confdConfig/logs/snmpLog - container element`
snmpLog is a log for tracing SNMP requests and responses. This log is disabled by default. In all other regards it can be configured as /confdConfig/logs/confdLog. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/snmpLogLevel` (error|info) [info]
Controls which level of SNMP pdus are printed in the SNMP log. The value "error" means that only PDUs with error-status not equal to "noError" are printed.

`/confdConfig/logs/netconfLog - container element`
netconfLog is a log for troubleshooting NETCONF operations, such as checking why e.g. a filter operation didn't return the data requested. This log is enabled by default. In all other regards it can be configured as /confdConfig/logs/confdLog. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/netconfTraceLog - container element`
netconfTraceLog is a log for understanding and troubleshooting NETCONF protocol interactions. When this log is enabled, all NETCONF traffic to and from ConfD is stored to a file. By default, all XML is pretty-printed. This will slow down the NETCONF server, so be careful when enabling this log. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/netconfTraceLog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", all NETCONF traffic is logged.

`/confdConfig/logs/netconfTraceLog/filename` (xs:string)
The name of the file where the NETCONF traffic trace log is written.

`/confdConfig/logs/netconfTraceLog/format` (pretty|raw) [pretty]
The value 'pretty' means that the XML data is pretty-printed. The value 'raw' means that it is not.

`/confdConfig/logs/xpathTraceLog - container element`
xpathTraceLog is a log for understanding and troubleshooting XPath evaluations. When this log is enabled, the execution of all XPath queries evaluated by ConfD is logged to a file.

This will slow down ConfD, so be careful when enabling this log. This log is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/xpathTraceLog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", all XPath execution is logged.

`/confdConfig/logs/xpathTraceLog/filename` (xs:string)
The name of the file where the XPath trace log is written.

`/confdConfig/logs/webuiAccessLog - container element`
webuiAccessLog is an access log for the embedded ConfD Web server. This file adheres to the Common Log Format, as defined by Apache and others. This log is not enabled by default and is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/trafficLog` (xs:boolean) [false]
Is either "true" or "false". If "true", all HTTP(S) traffic towards the embedded Web server is logged in a log file named "traffic.trace". Beware: Do not use this log in a production setting. This log is not enabled by default and is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/webuiAccessLog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the access log is used.

`/confdConfig/logs/webuiAccessLog/dir` (xs:string)
The path to the directory whereas the access log should be written to.

`/confdConfig/logs/webuiBrowserLog - container element`
webuiBrowserLog makes it possible to log Javascript errors/exceptions in a log file on the target device instead of just in the browser's error console. This log is not enabled by default and is not rotated, i.e. use logrotate(8).

`/confdConfig/logs/webuiBrowserLog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the browser log is used.

`/confdConfig/logs/webuiBrowserLog/filename` (xs:string)
The path to the filename whereas browser log entries should be written to.

`/confdConfig/logs/errorLog - container element`
errorLog is an error log used for internal logging from the ConfD daemon. It is used for troubleshooting the ConfD daemon itself, and should normally be disabled. This log is rotated by the ConfD daemon (see below).

`/confdConfig/logs/errorLog/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", error logging is performed.

`/confdConfig/logs/errorLog/filename` (xs:string)
filename is the full path to the actual log file. This parameter must be set if the errorLog is enabled.

`/confdConfig/logs/errorLog/maxSize` (confd:size) [S1M]
maxSize is the maximum size of an individual log file before it is rotated. Log filenames are reused when five logs have been exhausted.

`/confdConfig/datastores - container element`
datastores defines which datastores the ConfD daemon should be setup to handle.

`/confdConfig/datastores/startup - container element`
ConfD may keep separate running and startup configuration databases. When the system reboots for whatever reason, the running config database is lost, and the startup is read.

`/confdConfig/datastores/startup/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", a startup database is managed. Enable this only if our system uses a separate startup and running database.

`/confdConfig/datastores/candidate - container element`
ConfD may keep a shared, named alternative configuration database which can be modified without impacting the running configuration. Changes in the candidate can be commit to running, or discarded.

`/confdConfig/datastores/candidate/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", a candidate database is managed. Enable this if we want our users to use this feature from NETCONF, CLI or Webui, or other agents.

`/confdConfig/datastores/candidate/implementation` (confd|external) [confd]
implementation is either "confd" or "external". By default, ConfD implements the candidate configuration without impacting the application. But if our system already implements the candidate itself, set "implementation" to "external". This implies that the "external" candidate implementation must implement 5 C function callbacks for candidate manipulation. (See confd_lib_dp(3) and the example "misc/extern_candidate".

`/confdConfig/datastores/candidate/storage` (disk|ram|auto) [auto]
storage is either "disk", "ram", or "auto".

disk ConfD always stores the candidate on disk. "filename" must also be set (see below).

ram ConfD always stores the candidate in ram.

Note

If this alternative is used the implementation is not fully NETCONF compliant if confirmed-commit is enabled. The reason is that when confirmed-commit is used, the system must rollback to the previous configuration if it reboots but RAM data are lost.

auto ConfD stores the candidate on disk if absolutely necessary for proper operation; otherwise it is stored in ram. "filename" must also be set (see below).

`/confdConfig/datastores/candidate/filename` (xs:string)
filename is the name of the file where the candidate will be stored, if implementation is "confd" and "storage" is "disk" or "auto".

`/confdConfig/datastores/running - container element`
By default, the running configuration is writable. This means that the application must be prepared to handle dynamic changes to the configuration.

`/confdConfig/datastores/running/access` (read-write|writable-through-candidate) [read-write]
access is either "read-write" or "writable-through-candidate". If "writable-through-candidate", the candidate datastore must be enabled.

NOTE: The default value of read-write here is somewhat unfortunate. If the candidate is enabled it is better to set the value writable-through-candidate. When a NETCONF manager reconfigures a node that has the candidate and also read-write running, the manager can never know that running is up to date with the candidate and must thus always (logically) copy running to the candidate prior to modifying the candidate.

`/confdConfig/scripts`
It is possible to add scripts to control various things in ConfD, such as post-commit callbacks. New CLI commands can also be added. The scripts must be stored under /confdConfig/scripts/dir where there is a sub-directory for each script category. For some script categories it suffices to just add a script in the correct the sub-directory in order to enable the script. For others some configuration needs to be done.

`/confdConfig/scripts/dir`
The directory path to the location of plug-and-play scripts. The scripts directory must have the following sub-directories:

```
scripts/command/
        policy/
        post-commit/
```

`/confdConfig/defaultHandlingMode` (explicit|trim|report-all) [explicit]
defaultHandlingModeType is either "explicit", "trim", or "report-all". This parameter controls how default values for leafs are handled in the northbound interfaces.

explicit If a value is set over a northbound interface, it is not considered default.

Default values are not displayed in northbound interfaces, unless asked for by the user.

trim — If a value is set over a northbound interface, it is considered default if it is equal to the data model's default value.

Values equal to the data model's default value are not displayed in northbound interfaces, unless asked for by the user.

report-all — All values are logically stored in the data store, and displayed in northbound interfaces.

If the data store has the capability to handle default values, which for example CDB has, it will work with all values for this parameter. In this case, default values are never actaully stored in the data store. The value of this parameter should be chosen to give the end-user the best experience.

If the data store does not have the capability to handle default values, this parameter should be set to 'report-all'.

`/confdConfig/sortTransactions` (xs:boolean) [true]

This parameter controls how ConfD lists newly created, not yet committed list entries. If this value is set to 'false', ConfD will list all new entries before listing existing data. This was the only behavior in pre 2.5 versions of ConfD.

If this value is set to 'true', ConfD will merge new and existing entries, and provide one sorted view of the data. This behavior works well when CDB is used to store configuration data, but if an external data provider is used, ConfD does not know the sort order, and can thus not merge the new entries correctly. If an external data provider is used for configuration data, and the sort order differs from CDB's sort order, this parameter should be set to 'false'.

`/confdConfig/enableAttributes` (xs:boolean) [false]

This parameter controls if ConfD's attribute feature should be enabled or not. Currently there are three attributes, annotation, tags, and inactive. These are available in northbound interfaces (e.g. the annotate command in the CLI, and annotation XML attribute in NETCONF), but in order to be useful they need support from the underlying configuration data provider. CDB supports attributes, but if an external data provider is used for configuration data, and it does not support the attribute callbacks, this parameter should be set to 'false'.

`/confdConfig/enableInactive` (xs:boolean) [false]

This parameter controls if ConfD's inactive feature should be enabled or not. This feature also requires enableAttributes to be enabled.

`/confdConfig/sessionLimits - container element`

Parameters for limiting concurrent access to ConfD.

`/confdConfig/sessionLimits/maxSessions` (xs:unsignedInt|unbounded) [unbounded]

Puts a limit to the total number of concurrent sessions to ConfD.

`/confdConfig/sessionLimits/sessionLimit - container element`

Parameters for limiting concurrent access for a specific context to ConfD. There can be multiple instances of this container element, each one specifying parameters for a specific context.

`/confdConfig/sessionLimits/sessionLimit/context` (cli|netconf|xs:token)

The context is either one of cli, netconf, webui, snmp, rest, or it can be any other context string defined through the use of MAAPI. As an example, if we use MAAPI to implement a CORBA interface to ConfD, our MAAPI program could send the string "corba" as context.

`/confdConfig/sessionLimits/sessionLimit/maxSessions` (xs:unsignedInt|unbounded) [unbounded]
Puts a limit to the total number of concurrent sessions to ConfD for the corresponding context.

`/confdConfig/sessionLimits/maxConfigSessions` (xs:unsignedInt|unbounded) [unbounded]
Puts a limit to the total number of concurrent configuration sessions to ConfD.

`/confdConfig/sessionLimits/configSessionLimit - container element`
Parameters for limiting concurrent read-write transactions for a specific context to ConfD. There can be multiple instances of this container element, each one specifying parameters for a specific context.

`/confdConfig/sessionLimits/configSessionLimit/context` (cli|netconf|xs:token)
The context is either one of cli, netconf, webui, snmp, rest, or it can be any other context string defined through the use of MAAPI. As an example, if we use MAAPI to implement a CORBA interface to ConfD, our MAAPI program could send the string "corba" as context.

`/confdConfig/sessionLimits/configSessionLimit/maxSessions` (xs:unsignedInt|unbounded) [unbounded]
Puts a limit to the total number of concurrent configuration sessions to ConfD for the corresponding context.

`/confdConfig/capi - container element`
C-API parameters.

`/confdConfig/capi/newSessionTimeout` (xs:duration) [PT30S]
Timeout for a daemon to respond to a control socket request, see confd_lib_dp(3). If the daemon fails to respond within the given time, it will be disconnected.

`/confdConfig/capi/queryTimeout` (xs:duration) [PT120S]
Timeout for a daemon to respond to a worker socket query, see confd_lib_dp(3). If the daemon fails to respond within the given time, it will be disconnected.

`/confdConfig/capi/connectTimeout` (xs:duration) [PT60S]
Timeout for a daemon to send initial message after connecting the socket to the confd server. If the daemon fails to initiate the connection within the given time, it will be disconnected.

`/confdConfig/capi/objectCacheTimeout` (xs:duration) [PT2S]
Timeout for the cache used by the `get_object()` and `get_next_object()` callback requests, see confd_lib_dp(3). ConfD caches the result of these calls and serves get_elem requests from northbound agents from the cache.

Note

Setting this timeout too low will effectively cause the callbacks to be non-functional - e.g. `get_object()` may be invoked for each get_elem request from a northbound agent.

`/confdConfig/capi/eventReplyTimeout` (xs:duration) [PT120S]
Timeout for the reply from an event notification subscriber for a notification that requires a reply, see confd_lib_events(3). If the subscriber fails to reply within the given time, the notification socket will be closed.

`/confdConfig/rollback - container element`
Settings controlling if and where rollback files are created. A rollback file contains a copy of the system configuration. The current running configuration is always stored in rollback0, the previous version in rollback1 etc. The oldest saved configuration has the highest suffix.

`/confdConfig/rollback/enabled` (xs:boolean) [false]
If "true", then a rollback file will be created whenever the running configuration is modified. If /confdConfig/ha/enabled is set to true then rollback files are replicated from master to slaves. The rollback feature must be enabled on all HA nodes (although is possible to have a slave with rollbacks disabled, that slave will then not replicate the rollback files).

`/confdConfig/rollback/directory` (xs:string)
Location where rollback files will be created.

`/confdConfig/rollback/historySize` (xs:unsignedInt) [50]
Number of old configurations to save.

`/confdConfig/rollback/type` (full|delta) [delta]
Type of rollback file to use. If "full" is specified, then a full configuration dump is stored in each rollback file. Rollback file 0 will always contain the running configuration.

If "delta" is used, then only the changes are stored in the rollback file. Rollback file 0 will contain the changes from the last configuration.

Using deltas is more space and time efficient for large configurations. Full rollback files are more robust when multiple external databases are used. If the external databases becomes inconsistent a previous configuration can always be restored using a full rollback file.

`/confdConfig/rollback/rollbackNumbering` (fixed|rolling) [rolling]
rollbackNumbering is either "fixed" or "rolling". If set to "rolling" then rollback file "0" will always contain the last commit. When using "fixed" each rollback will get a unique increasing number.

`/confdConfig/ssh - container element`
This section defines settings which affect the behavior of the builtin SSH implementation.

`/confdConfig/ssh/idleConnectionTimeout` (xs:duration) [PT10M]
The maximum time that an authenticated connection to the SSH server is allowed to exist without open channels. If the timeout is reached, the SSH server closes the connection. Default is PT10M, i.e. 10 minutes. If the value is 0, there is no timeout.

`/confdConfig/ssh/algorithms - container element`
This section defines custom lists of algorithms to be usable with the built-in SSH implementation.

For each type of algorithm, an empty value means that all supported algorithms should be usable, and a non-empty value (a comma-separated list of algorithm names) means that the intersection of the supported algorithms and the configured algorithms should be usable.

`/confdConfig/ssh/algorithms/serverHostKey` (xs:string)
The supported serverHostKey algorithms (if implemented in libcrypto) are "ssh-dss" and "ssh-rsa", but for any SSH server, it is limited to those algorithms for which there is a host key installed in the directory given by /confdConfig/aaa/sshServerKeyDir.

To limit the usable serverHostKey algorithms to "ssh-dss", set this value to "ssh-dss" or avoid installing a key of any other type than ssh-dss in the sshServerKeyDir.

`/confdConfig/ssh/algorithms/kex` (xs:string)
The supported key exchange algorithms (as long as their hash functions are implemented in libcrypto) are "diffie-hellman-group-exchange-sha256", "diffie-hellman-group-exchange-sha1", "diffie-hellman-group14-sha1" and "diffie-hellman-group1-sha1".

To limit the usable key exchange algorithms to "diffie-hellman-group14-sha1" and "diffie-hellman-group-exchange-sha256" (in that order) set this value to "diffie-hellman-group14-sha1,diffie-hellman-group-exchange-sha256".

`/confdConfig/ssh/algorithms/mac` (xs:string)
The supported mac algorithms (if implemented in libcrypto) are "hmac-md5", "hmac-sha1", "hmac-sha2-256", "hmac-sha2-512", "hmac-sha1-96" and "hmac-md5-96".

`/confdConfig/ssh/algorithms/encryption` (xs:string)
The supported encryption algorithms (if implemented in libcrypto) are "aes128-ctr", "aes192-ctr", "aes256-ctr", "aes128-cbc", "aes256-cbc" and "3des-cbc".

`/confdConfig/ssh/clientAliveInterval` (xs:duration|infinity) [infinity]
If no data has been received from a connected client for this long, a request that requires a response from the client, will be sent over the SSH transport.

`/confdConfig/ssh/clientAliveCountMax` (xs:unsignedInt) [3]
If no data has been received from the client, after this many consecutive clientAliveInterval has passed, the connection will be dropped.

`/confdConfig/cli - container element`
CLI parameters.

`/confdConfig/cli/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the CLI server is started.

`/confdConfig/cli/transactions` (xs:boolean) [true]
Control whether commit should be supported in the CLI or not. When set to false all commands will be automatically committed when the user press ENTER in the CLI.

`/confdConfig/cli/transactionCntrlCmds` (xs:boolean) [false]
transactionCtrlCmds is either "true" or "false". If "true", then the CLI will have commands for enabling and disabling transactions in configure mode, ie "enable transactions" and "disable transactions". If set to "false" no such commands will be present.

`/confdConfig/cli/startupScriptNonInteractive` (xs:boolean) [false]
startupScriptNonInteractive is either "true" or "false". If set to "true" then a CLI startup script will be evaluated also for non-interactive sessions.

`/confdConfig/cli/tableLabel` (xs:boolean) [false]
tableLabel is either "true" or "false". If "true" then tables displayed in C and I style CLI will have a relative location label to make it possible to know which table is displayed.

`/confdConfig/cli/addExtraTableSpacing` (xs:boolean) [false]
addExtraTableSpacing is either "true" or "false". If set to "true" then an additional newline will be added on each side of the table.

`/confdConfig/cli/tableLookAhead` (xs:positiveInteger) [50]
The tableLookAhead element tells confd how many rows to pre-fetch when displaying a table. The prefetched rows are used for calculating the required column widths for the table. If set to a small number it is recommended to explicitly configure the column widths in the clispec file.

`/confdConfig/cli/moreBufferLines` (xs:unsignedInt|unbounded) [5000]
moreBufferLines is used to limit the buffering done by the more process. It can be "unbounded" or a positive integer describing the maximum number of lines to buffer.

`/confdConfig/cli/showTableLabelsIfMultiple` (xs:boolean) [false]
showTableLabelsIfMultiple is either "true" or "false". If set to "true" then table labels will only be displayed if multiple tables, or a table and additional data is displayed. If set to "false" then table labels are always shown if they exists and tableLabel is enabled.

`/confdConfig/cli/editWrapMode` (wrap|newline|vt100) [wrap]
editWrapMode is either "wrap", "newline" or "vt100". If "wrap" or "vt100" is used then cut-and-paste will work in xterms (and other terminal emulators) but the CLI may behave oddly if the screen width is manually configured to something other than the true screen width. If "vt100" is used then no >space<>backspace< is used to force line wrapping. This makes it easier for scripts that rely on command line echoing but the cursor will disappear at the end of the line temporarily.

`/confdConfig/cli/supportQuoteEOL` (xs:boolean) [true]
supportQuoteEOL is either true or false. If set to true then a final backslash (\) on a line means that the next line will be concatenated with the previous line, similarly to a Linux Shell.

`/confdConfig/cli/useShortEnabled` (xs:boolean) [true]
useShortEnabled is either "true" or "false". If set to "true" then the CLI will display "enabled" or "disabled" in place of "enabled true" and "enabled false".

`/confdConfig/cli/smartRenameFiltering` (xs:boolean) [true]
smartRenameFiltering is either "true" or "false". If set to "true" then only paths that leads to existing instances will be presented when doing completion. This will lead to some extra calls to get_next() in order to determine if a path has instances or not. When set to "false" all paths with potentially instances are presented.

`/confdConfig/cli/allowWildcard` (xs:boolean) [true]
allowWildcard is either "true" or "false". If "true" then wildcard expressions are allowed in show commands.

`/confdConfig/cli/allowAllAsWildcard` (xs:boolean) [false]
allowAllAsWildcard is either "true" or "false". If "true" then "all" can be used in place of "*" as wildcard.

`/confdConfig/cli/allowRangeExpression` (xs:boolean) [true]
allowRangeExpression is either "true" or "false". If "true" then range expressions are allowed for all key values of type basic type integer. An alternative is to specify hasRange for each path in the clispec.

`/confdConfig/cli/allowRangeExpressionAllTypes` (xs:boolean) [true]
allowRangeExpressionAllTypes is either "true" or "false". If "true" then range expressions are allowed for all key values regardless of type. An alternative is to specify hasRange for each element in the yang files.

`/confdConfig/cli/suppressRangeKeyword` (xs:boolean) [false]
suppressRangeKeyword is either "true" or "false". If "true" then 'range' keyword is not allowed in C- and I-style for range expressions.

`/confdConfig/cli/exitModeOnEmptyRange` (xs:boolean) [false]
exitModeOnEmptyRange is either "true" or "false". If "true" and if standing in a range submode, the CLI will exit to the parent submode if all instances in the range has been deleted by the user.

`/confdConfig/cli/useDoubleDotRanges` (xs:boolean) [false]
useDoubleDotRanges is either "true" or "false". If "true" then range expressions are types as 1..3, if set to "false" then ranges are given as 1-3.

`/confdConfig/cli/singleElemPattern` (xs:boolean) [true]
singleElemPattern is either "true" or "false". If "true" then filters/patterns to show commands can be used to specify that you want to see a specific leaf element of all lists. Only that leaf element will be shown for each list entry. This works for both tables and row based rendering. To get the old 2.7 behavior set this flag to "false".

`/confdConfig/cli/multiPatternOperation` (any/all) [any]
multiPatternOperation is one of "any", and "all". When set to "any" a pattern is true if at least one match is found, if set to "all", all patterns needs to be found for the pattern expression to be considered true.

`/confdConfig/cli/defaultTableBehavior` (dynamic|enforce|suppress) [dynamic]
defaultTableBehavior is either "dynamic", "suppress", or "enforce". If set to "dynamic" then list nodes will be displayed as tables if the resulting table will fit on the screen. If set to suppress, then list nodes will not be displayed as tables unless a table has been specified by some other means (ie through a setting in the clispec-file or through a command line parameter), if set to "enforce" then list nodes will always be displayed as tables unless otherwise specified in the clispec-file or on the command line.

`/confdConfig/cli/allowTableOverflow` (xs:boolean) [false]
allowTableOverflow is either "true" or "false". If "true" then tables displayed in a Cisco style CLI will be allowed to overflow. If "false" a too wide table will be displayed as a "setting - value" list instead.

`/confdConfig/cli/tableOverflowTruncate` (xs:boolean) [false]
tableOverflowTruncate is either "true" or "false". If "true" then overflowing tables will be truncated instead of wrapped.

`/confdConfig/cli/allowTableCellWrap` (xs:boolean) [true]
allowTableCellWrap is either "true" or "false". If "true" then tables displayed in a Cisco style CLI will be allowed to wrap if the initial cell-width estimate proves to be too narrow. If "false" a too wide table cell will overflow instead, pushing the rest of the line to the right.

`/confdConfig/cli/compactTable` (xs:boolean) [false]
compactTable is either "true" or "false". If "true" then tables with multiple dynamic levels will be displayed more compactly. The first instance of the sub-element will appear on the same row as the parent instance. When set to "false" all new instances will appear on a new row.

`/confdConfig/cli/showKeyName` (xs:boolean) [true]
showKeyName is either "true" or "false". If "true" then the key name will be displayed in the completion list during completion.

`/confdConfig/cli/promptEnumLimit` (xs:nonNegativeInteger) [4]
promptEnumLimit controls how many enumerations should be included in the prompt when prompting the user for a value where there are a number of alternatives. If the number of alternatives exceeds the above configured limit then the list will be truncated and the string "..." will be added.

`/confdConfig/cli/showAllNs` (xs:boolean) [false]
showAllNs is either "true" or "false". If "true" then all element names will be shown with their namespace prefix in the CLI. This is visible when displaying the running configuration and when modifying the configuration.

`/confdConfig/cli/useExposeNsPrefix` (xs:boolean) [true]
useExposeNsPrefix is either "true" or "false". If "true" then all nodes annotated with the tailf:cli-expose-ns-prefix will result in the namespace prefix being shown/required. If set to "false" then the tailf:cli-expose-ns-prefix annotation will be ignored.

`/confdConfig/cli/sortSubmodeCmds` (xs:boolean) [true]
sortSubmodeCmds is either "true" or "false". If set to "true" then local submode commands are listed before global commands when the user enters ? in a submode in C and I-style.

`/confdConfig/cli/sortLocalCmds` (xs:boolean) [true]
sortLocalCms is either "true" or "false". If set to "true" and sortSubmodeCmds are also set to true, then local submode commands are listed before global commands when the user enters ? in a submode

in C and I-style, and the order of the commands is alphabetically ordered. If set to false then the order of the local submode commands are the same as in the data model.

`/confdConfig/cli/allowOldStyleModeCmds` (xs:boolean) [false]
allowOldStyleModeCmds is either "true" or "false". If set to "true" then CLI commands in I and C-style are interpreted as mode commands if the path coincide with a list in the data-model. The recommended way to mount commands in a submode is instead to use the "mount" attribute.

`/confdConfig/cli/continueOnErrorCmdStack` (xs:boolean) [false]
continueOnErrorCmdStack is either "true" or "false". If set to "true" then command stack execution will continue even if an earlier command in the stack failed with an error, ie show xx ; show zz will execute both 'show xx' and 'show zz' even if 'show xx' failed with an error. If set to "false" then command execution will be aborted as soon as a command fails.

`/confdConfig/cli/completionShowOldVal` (xs:boolean) [true]
completionShowOldVal is either "true" or "false". If set to "true" a leaf's old value will be displayed inside brackets during command line completion. If set to "false" it will not be shown.

`/confdConfig/cli/completionMetaInfo` (false|alt1|alt2) [false]
completionMetaInfo is either "false", "alt1" or "alt2". This option only applies to the J-style CLI. If set to "alt1" then the alternatives shown for possible completions will be prefixed as follows: containers with >, lists with +, and leaf-lists +. For example:

```
Possible completions:
              ...
              > applications
              + apply-groups
              ...
              + dns-servers
              ...
```

If set to "alt2", then possible completions will be prefixed as follows: containers with >, lists with children with +>, and lists without children +. For example:

```
              Possible completions:
              ...
              > applications
              +>apply-groups
              ...
              + dns-servers
              ...
```

`/confdConfig/cli/reportInvalidCompletionInput` (xs:boolean) [true]
reportInvalidCompletionInput is either true or false. If set to 'true' the CLI will display an error message during completion when the user press '?' to indicate if an invalid token has been entered on the command line.

`/confdConfig/cli/banner` (xs:string)
Banner shown to the user when the CLI is started. Default is empty.

`/confdConfig/cli/bannerFile` (xs:string)
Name of file whose contents are shown to the user when the CLI is started. If empty, the message, if any, set via /confdConfig/cli/banner is shown. Default is empty.

`/confdConfig/cli/completionShowMax` (xs:integer) [100]
Maximum number of possible alternatives to present when doing completion.

`/confdConfig/cli/rollbackAAA` (xs:boolean) [false]
If set to true then AAA rules will be applied when a rollback file is loaded. This means that rollback may not be possible if some other user have made changes that the current user does not have access privileges to.

`/confdConfig/cli/rollbackMax` (xs:integer)
Maximum number of rollback changes to allow through the CLI

`/confdConfig/cli/rollbackNumbering` (fixed|rolling) [rolling]
rollbackNumbering is either "fixed" or "rolling". If set to "rolling" then rollback file "0" will always contain the last commit. When using "fixed" each rollback will get a unique increasing number.

`/confdConfig/cli/rollbackNumberingInitial` (xs:integer) [10000]
rollbackNumberingInitial is the starting point of the rollback numbering when the "increasing" rollback numbering scheme is used.

`/confdConfig/cli/historyMaxSize` (xs:integer) [1000]
Sets maximum configurable history size.

`/confdConfig/cli/messageMaxSize` (xs:integer) [10000]
Maximum size of user message.

`/confdConfig/cli/compactShow` (xs:boolean) [false]
Use compact representation when showing the configuration in C and I style CLIs.

`/confdConfig/cli/compactStatsShow` (xs:boolean) [false]
Use compact representation when showing the operational data in C and I style CLIs.

`/confdConfig/cli/prettifyStatsName` (xs:boolean) [false]
Default setting for prettifying, ie changing _ and - to space in element names when displaying config='false' data in key-value listings.

`/confdConfig/cli/displayEmptyConfigContainers` (xs:boolean) [true]
displayEmptyConfigContainers is either "true" or "false". If set to "true" then 'show status' in the J-style CLI will display empty lists that are "config true" even when there is no data to display. If set to "false" the those containers will not be shown.

`/confdConfig/cli/confirmUncommitedOnExit` (prompt|discard) [prompt]
If set to 'prompt' then the user will be prompted whether to discard uncommitted changes or not. If set to 'discard' then uncommitted changes will be discarded without prompting the user. If set to 'commit' then uncommitted changes will be automatically committed without asking the user.

`/confdConfig/cli/reconfirmHidden` (xs:boolean) [false]
If set to true the user will have to re-confirm non-echoing values in the CLI. Ie, when the CLI prompts the user for a value that is not echoed the user will be asked to enter it twice.

`/confdConfig/cli/dequoteHidden` (xs:boolean) [false]
If set to true the value that the user entered will be unquoted, ie if the user enters \n it will be interpreted as a newline. This is the default behavior for all other leaf types. If set to false then no unquoting will be performed for hidden (non-echoing) data types when the CLI explicitly prompts for their values. Dequoting will still be performed for values entered directly on the command line.

`/confdConfig/cli/enumKeyInfo` (xs:boolean) [false]
If set to true the CLI will add the text <keyname:enumeration> whenever it is displaying a completion list for entering a key value that is an enumeration. For example:

```
io(config)# vqe dsp channel 1
Possible completions:
<b-id:enumeration>  10  11  12  13  14  5  6  9
```

`/confdConfig/cli/historyRemoveDuplicates` (xs:boolean) [false]
If set to "true" then repeated commands in the CLI will only be stored once in the history. Each invocation of the command will only update the date of the last entry. If set to "false" duplicates will be stored in the history.

`/confdConfig/cli/unifiedHistory` (xs:boolean) [false]
If set to "true" then the 'show history' command will display the unified command history, ie the command history from all modes. If set to "false" then only the command history from the current mode will be shown.

`/confdConfig/cli/showDefaults` (xs:boolean) [false]
showDefaults is either "true" or "false". If "true" then default values will be shown when displaying the configuration. The default value is shown inside a comment on the same line as the value. Showing default values can also be enabled in the CLI per session using the operational mode command "set show defaults true".

`/confdConfig/cli/reallocateOperTrans` (xs:boolean) [false]
reallocateOperTrans is either "true" or "false". If "true" then a new read transaction will be allocated for each oper-mode command. When set to "false" a single oper transaction will be used for the entire CLI session.

`/confdConfig/cli/quickSshTeardown` (xs:boolean) [false]
quickSshTeardown controls if CLI sessions initiated through an SSH sessions should be torn down directly when the socket is closed, or not. When set to 'true' the socket will be closed as soon as the CLI receives a tcp shutdown, if set to 'false' it will wait until all pending data has been written.

`/confdConfig/cli/cAlignLeafValues` (xs:boolean) [true]
cAlignLeafValues is either "true" or "false". If "true" then the leaf values of all siblings in a container or list will be aligned.

`/confdConfig/cli/jAlignLeafValues` (xs:boolean) [true]
jAlignLeafValues is either "true" or "false". If "true" then the leaf values of all siblings in a container or list will be aligned.

`/confdConfig/cli/columnStats` (xs:boolean) [false]
columnStats is either "true" or "false". If "false" then the container element is repeated on each line when displaying config="false" data in the C and I style CLIs using the "show" command. If set to "true" then the name of the container will not be repeated, instead all leaves will be indented.

`/confdConfig/cli/allowAbbrevCmds` (xs:boolean) [true]
allowAbbrevCmds is either "true" or "false". If "false" then commands are not allowed to be abbreviated in the CLI.

`/confdConfig/cli/allowAbbrevCmdsOnLoad` (xs:boolean) [true]
allowAbbrevCmdsOnLoad is either "true" or "false". If "false" then commands are not allowed to be abbreviated in the CLI in non interactive mode, ie when loading configurations from file.

`/confdConfig/cli/strictRefsOnLoad` (xs:boolean) [false]
strictRefsOnLoad is either "true" or "false". If "false" then keyref/leafref targets does not have to exist when loading a config from a file. If set to "true" then the target creation must appear earlier in the loaded file than the reference to the target. Note that there is a rather heavy performance penalty for loading files with many keyrefs when this is set to true, or for piping CLI commands into confd_cli.

`/confdConfig/cli/allowAbbrevKeys` (xs:boolean) [true]
allowAbbrevKeys is either "true" or "false". If "false" then key elements are not allowed to be abbreviated in the CLI. This is relevant in the J-style CLI when using the commands 'delete' and 'edit'. In the C/I-style CLIs when using the commands 'no', 'show configuration' and for commands to enter submodes.

`/confdConfig/cli/allowAbbrevParamNames` (xs:boolean) [false]
allowAbbrevParamNames is either "true" or "false". If "false" then cli command parameter names, ie <name>xx</name>, cannot be abbreviated.

`/confdConfig/cli/allowAbbrevEnums` (xs:boolean) [true]
allowAbbrevEnums is either "true" or "false". If "false" then enums entered in the cli cannot be abbreviated.

`/confdConfig/cli/allowCaseInsensitiveEnums` (xs:boolean) [true]
allowCaseInsensitiveEnums is either "true" or "false". If "false" then enums entered in the cli must match in case, ie you cannot enter FALSE if the cli asks for 'true' or 'false'.

`/confdConfig/cli/caseInsensitive` (xs:boolean) [false]
caseInsensitive is either "true" or "false". If "false" then all CLI commands must have the correct case. If set to "true" then case is mostly ignored. Note that if set to "true" then all data model files and clispec-files must be written with this in mind. You cannot have two elems that conflict in case.

`/confdConfig/cli/caseInsensitiveKeys` (xs:boolean) [false]
caseInsensitiveKeys is either "true" or "false". If "false" then all user defined instance names must have correct case. If set to "true" then case is mostly ignored. Note that if set to "true" then all data model files and clispec-files must be written with this in mind. You cannot have two elems that conflict in case.

`/confdConfig/cli/ignoreLeadingWhitespace` (xs:boolean) [false]
ignoreLeadingWhitespace is either "true" or "false". If "false" then the CLI will show completion help when the user enters TAB or SPACE as the first characters on a row. If set to "true" then leading SPACE and TAB are ignored. The user can enter '?' to get a list of possible alternatives. Setting the value to "true" makes it easier to paste scripts into the CLI.

`/confdConfig/cli/indentTemplates` (xs:boolean) [false]
indentTemplates is either "true" or "false". If set to "true" then the text resulting from a show template will be indented to the same level as the surrounding auto-rendered show text. If set to "false" then no automatic indentation will occur. The automatic variable ".indent" may be used in the templates to do manual indentation.

`/confdConfig/cli/compListCompact` (xs:boolean) [false]
compListCompact is either "true" or "false". If "true" then the CLI will display items with an associated info text one per line, and all the rest in compact format.

`/confdConfig/cli/completionListLine` (xs:boolean) [false]
completionListLine is either "true" or "false". If "true" then the CLI will display completion lists one item per line. If set to "false" one-line presentation will be used for items with info texts and compact for the rest. (if compListCompact above is set to true there may be a mix of the to formats in the same listing)

`/confdConfig/cli/showMatchBeforePossible` (xs:boolean) [false]
showMatchBeforePossible is either "true" or "false". If set to "true" then the match completions will be displayed before the other possible completions, if set to "false" then the match completions will be displayed after.

`/confdConfig/cli/wrapInfo` (xs:boolean) [true]
wrapInfo is either "true" or "false". If "false" then the CLI will not automatically wrap the info field in "Possible completion:" listings. If set to "true" then the info text will be word-wrapped and indented.

`/confdConfig/cli/wrapPrompt` (xs:boolean) [false]
wrapPrompt is either "true" or "false". If "false" then the CLI will not automatically word wrap the prompt when prompting the user for some input. If set to "true" then the prompt will be word-wrapped according to the current terminal width.

`/confdConfig/cli/sortShowElems` (xs:boolean) [true]
sortShowElems is either "true" or "false". If "false" then the show commands will display the elements in the order they appear in the data model. If set to "true" then all non-lists will appear before the lists. This setting only applies to the C- and I-style CLIs.

`/confdConfig/cli/possibleCompletionsFormat` (xs:string) [Possible completions]
possibleCompletionsFormat is the string displayed before the displaying the actual completion possibilities.

`/confdConfig/cli/matchCompletionsFormat` (xs:string) [Possible match completions:]
matchCompletionsFormat is the string displayed before the displaying the actual match completion possibilities.

`/confdConfig/cli/noMatchCompletionsFormat` (xs:string) [Possible match completions:]
noMatchCompletionsFormat is the string displayed when there are no matching completion possibilities. The string is empty by default.

`/confdConfig/cli/showDescription` (xs:boolean) [true]
showDescription is either true or false. If set to false then the Description: xx text will not be displayed.

`/confdConfig/cli/explicitSetCreate` (xs:boolean) [false]
explicitSetCreate is either "true" or "false". If set to "true" then the 'set' command in J-style CLI cannot be used to create instances. Instead a new command called 'create' becomes available for creating instances. Note that this deviates from a typical Juniper style CLI where instance creation is performed by the 'set' command.

`/confdConfig/cli/templateFilter - container element`
User defined template filters must be listed here. They can be used in show templates in the same manner as the builtin ones. A template filter takes a string as input in 'argv[1]' and returns a modified version of it by invoking 'confd_action_reply_command'. It can also take extra arguments. For example '|a_filter:foo:42' implies 'argv[2]="foo"' and 'argv[3]="42".

`/confdConfig/cli/templateFilter/name` (string)
Name of template filter.

`/confdConfig/cli/templateFilter/callback` (string)
Name of callback. The callback receives a string as first argument, optionally followed by the list of arguments given to the filter in the show template.

`/confdConfig/cli/enableDisplayLevel` (true|false|pipe) [pipe]
enableDisplayLevel is either "true", "false" or "pipe". If "false" then the 'displaylevel' option to the show command will not be available in the CLIs. If set to "pipe" then a special pipe target called 'display-level' will be available.

The displaylevel option can be used to limit how many levels will be displayed by the show command. If a display level of 1 is specified then only the direct children of an element will be shown. If a display level of 3 is specified then only elements at depth 3 below a given element will be displayed, etc.

A user can also modify the default display level for a given CLI- session using the display-level setting in the CLI, similarly to the screen-width, or idle-timeout settings.

`/confdConfig/cli/enableDisplayGroups` (xs:boolean) [true]
enableDisplayGroups is either "true" or "false". If "false" then the user will not be able to provide a set of display groups when issuing the show command.

`/confdConfig/cli/defaultDisplayLevel` (xs:integer) [99999999]
If enableDisplayLevel is set to "true" then this settings controls the default display level used if no explicit display level is specified. It is also used as the initial value of the (set) 'display-level' command in the CLI.

`/confdConfig/cli/instanceDescription` (xs:boolean) [true]
instanceDescription is either "true" or "false". If "true" then the CLI will look for description elems and add their values as info texts when displaying possible completions in the CLI. This makes it easier to identify the different instances.

`/confdConfig/cli/addErrorPrefixSuffix` (xs:boolean) [true]
addErrorPrefixSuffix is either "true" or "false". If "true" then the CLI will add "Error: " or "Aborted: " and when operations fail in the CLI. If set to "false" then the prefix will not be added for errors generated by some callback.

`/confdConfig/cli/autocommitLoad` (xs:boolean) [false]
autocommitLoad is either "true" or "false". If "true" then when executing the 'load' command each line will be committed as soon as it has been read. Note that this is normally not a good idea. Only applies when transactions are disabled.

`/confdConfig/cli/autocommitLoadChunkSize` (xs:positiveInteger) [1]
autocommitLoadChunkSize is used to avoid auto commit:ing after each line but instead commit after a chunk of lines have been read.

`/confdConfig/cli/allOrNothingLoad` (xs:boolean) [false]
allOrNothingLoad is either true or false. If set to true then the transaction will be reset and all changes discarded if an error is encountered when loading a file. This behavior will not happen when the 'best effort' pipe target is used, nor when stopLoadOnError is set to false, nor when autocommitLoad is set to true.

`/confdConfig/cli/stopLoadOnError` (xs:boolean) [true]
stopLoadOnError is either "true" or "false". If "false" then the 'load' command in the C and I-style CLIs will not terminate on the first error but continue to process commands form the file.

`/confdConfig/cli/enableLoadMerge` (xs:boolean) [true]
enableLoadMerge is either "true" or "false". If "false" then the 'load' command in the C and I-style CLIs will not have an option for how to load a config file. If set to "true" then the 'load' command will have an additional option for loading the file either in 'override' mode or in 'merge' mode. 'override' is the mode used if enableLoadMerge is set to 'false'.

`/confdConfig/cli/oldDetailsArg` (xs:boolean) [false]
oldDetailsArg is either "true" or "false". If "false" then commands that display the configuration will not have a "details" argument but instead have a pipe flag called "details". The setting is present for backwards compatibility, the recommended setting for future use is "false".

`/confdConfig/cli/withDefaults` (xs:boolean) [false]
DEPRECATED - use `/confdConfig/defaultHandlingMode` instead to control this behavior consistently for all northbound interfaces. Set `/confdConfig/defaultHandlingMode` to *`report-all`* to display default values.

withDefaults is either "true" or "false". If "false" then leaf nodes that have their default values will not be shown when the user displays the configuration, unless the user gives the "details" option to the "show" command.

This is useful when there are many settings which are seldom used. When set to "false" only the values actually modified by the user will be shown.

`/confdConfig/cli/ignoreShowWithDefaultOnDiff` (xs:boolean) [false]
When set to 'true' ConfD will ignore the annotation tailf:cli-show-with-default when displaying the configuration changes in the C-style CLI.

`/confdConfig/cli/trimDefaultShow` (xs:boolean) [false]
trimDefaultShow is either "true" or "false". If "true" then leaf nodes that have the same value as the default value will not be displayed even when explicitly configured to have that value. When set to "false" such leaves will be displayed if explicitly configured to have the value. This setting applies to show commands, ie show running-config and show config.

If this behavior is wanted, it is recommended to set `/confdConfig/defaultHandlingMode` to *`trim`* instead of using this parameter, in order to get a consistent behavior for all northbound interfaces.

If the default handling mode is *`trim`*, explicitly configured values that are the same as the default value are never stored in the data store. This means that if the default handling mode is *`trim`*, this parameter has no effect.

`/confdConfig/cli/trimDefaultSave` (xs:boolean) [false]
trimDefaultSave is either "true" or "false". If "true" then leaf nodes that have the same value as the default value will not be displayed even when explicitly configured to have that value. When set to "false" such leaves will be displayed if explicitly configured to have the value. This setting applies to the save command.

If this behavior is wanted, it is recommended to set `/confdConfig/defaultHandlingMode` to *`trim`* instead of using this parameter, in order to get a consistent behavior for all northbound interfaces.

If the default handling mode is *`trim`*, explicitly configured values that are the same as the default value are never stored in the data store. This means that if the default handling mode is *`trim`*, this parameter has no effect.

`/confdConfig/cli/docWrap` (xs:boolean) [true]
docWrap is either "true" or "false". If "false" then certain documentation texts will not be enclosed in "<" and ">", if set to "true" they will be.

`/confdConfig/cli/infoOnMatch` (xs:boolean) [true]
infoOnMatch is either "true" or "false". If "true" then the CLI will add info texts when displaying possible match completions. If set to "false" then the info text will not be shown.

`/confdConfig/cli/externalActionErrorMsg` (xs:string)
The externalActionErrorMsg value is displayed whenever an external error occurs when executing an action in the CLI.

`/confdConfig/cli/infoOnTab` (xs:boolean) [true]
infoOnTab is either "true" or "false". If "false" then no info strings will be displayed in the tab completion list when the user enters TAB.

`/confdConfig/cli/infoOnSpace` (xs:boolean) [true]
infoOnSpace is either "true" or "false". If "false" then no info strings will be displayed in the tab completion list when the user enters SPACE.

`/confdConfig/cli/newLogout` (xs:boolean) [true]
newLogout is either "true" or "false". If "false" then the I and C modes will have a single "logout" command for logging out a user and a specific session. If set to "true" then there will be two different commands - "logout user <name>" and "logout session <id>"

`/confdConfig/cli/newInsert` (xs:boolean) [true]
newInsert is either "true" or "false". If "false" then the old insert command will be used. If set to "true" then the new insert command, capable of inserting ordered-by-user list elements, will be used.

`/confdConfig/cli/showEditors` (xs:boolean) [true]
showEditors is either true or false. If set to true then a list of current editors will be displayed when a user enters configure mode.

`/confdConfig/cli/whoShowMode` (xs:boolean) [true]
whoShowMode is either "true" or "false". If set to "true" then an 'Config Mode' column will be added to the table shown when issuing the 'who' command in C- and I-mode.

`/confdConfig/cli/whoHistoryDateTimeFormat` (short|long) [short]
whoHistoryDateTimeFormat decides if the date should always include the date (long), or only include the date when different from today (short).

`/confdConfig/cli/messageFormat` (xs:string) [Message from $(sender) at $(time)...\n $(message)\nEOF\n]
messageFormat controls how messages between users and from the system should be presented to the user. The format string may contain the variables $(sender), $(time), $(message), $(date), $(time12), $(time12ampm), $(time12hm), $(host), $(hostname), and $(user).

`/confdConfig/cli/messageWordWrap` (xs:boolean) [false]
messageWordWrap is either "true" or "false". If set to "true" then all system/user/prio messages in the CLI will be word-wrapped to the current terminal width.

`/confdConfig/cli/messageQueueSize` (xs:integer) [10]
Some messages are not displayed in the CLI when a command executed, but are delayed until the current command execution has finished. The size of the queue of pending messages is configured in messageQueueSize.

`/confdConfig/cli/defaultPrefix` (xs:string) []
defaultPrefix is a string that is placed in front of the default value when a configuration is shown with default values as comments.

`/confdConfig/cli/jWarningPrefix` (xs:string) [Warning:]
jWarningPrefix is a string that is placed in front of warnings when they are displayed in the CLI. J-style CLI.

`/confdConfig/cli/jAbortedPrefix` (xs:string) [Aborted:]
jAbortedPrefix is a string that is placed in front of aborted messages when they are displayed in the CLI. J-style CLI.

`/confdConfig/cli/jErrorPrefix` (xs:string) [Error:]
jErrorPrefix is a string that is placed in front of error messages when they are displayed in the CLI. J-style CLI.

`/confdConfig/cli/cWarningPrefix` (xs:string) [Warning:]
cWarningPrefix is a string that is placed in front of warnings when they are displayed in the CLI. I- and C-style CLI.

`/confdConfig/cli/cAbortedPrefix` (xs:string) [Aborted:]
cAbortedPrefix is a string that is placed in front of aborted messages when they are displayed in the CLI. I- and C-style CLI.

`/confdConfig/cli/cErrorPrefix` (xs:string) [Error:]
cErrorPrefix is a string that is placed in front of error messages when they are displayed in the CLI. I- and C-style CLI.

`/confdConfig/cli/invalidDataString` (xs:string) [--ERROR--]
invalidDataString is a string that is displayed instead of the real value whenever a data provider returns an invalid data element.

`/confdConfig/cli/prompt1` (xs:string) [\u@\h>]
Prompt used in operational mode. The string may contain a number of backslash-escaped special characters which are decoded as follows:

`\d`	the date in "YYYY-MM-DD" format (e.g., "2006-01-18")
`\h`	the hostname up to the first '.'
`\H`	the hostname
`\s`	the client source ip
`\S`	the name provided by the -H argument to confd_cli
`\t`	the current time in 24-hour HH:MM:SS format
`\T`	the current time in 12-hour HH:MM:SS format
`\@`	the current time in 12-hour am/pm format
`\A`	the current time in 24-hour HH:MM format
`\u`	the username of the current user
`\m`	the mode name (only used in XR style)
`\m{N}`	same as \m, but the number of trailing components in the displayed path is limited to be max N (an integer). Characters removed are replaced with an ellipsis (...).
`\M`	the mode name inside parenthesis if in a mode
`\M{N}`	same as \M, but the number of trailing components in the displayed path is limited to be max N (an integer). Characters removed are replaced with an ellipsis (...).

`/confdConfig/cli/prompt2` (xs:string) [\u@\h%]
Prompt used in configuration mode. The string may contain a number of backslash-escaped special characters which are decoded as described above.

`/confdConfig/cli/cPrompt1` (xs:string) [\h#]
Prompt used in operational mode in C style. The string may contain a number of backslash-escaped special characters which are decoded as described above.

`/confdConfig/cli/cPrompt2` (xs:string) [\h(\m)#]
Prompt used in configuration mode in C style. The string may contain a number of backslash-escaped special characters which are decoded as described above.

`/confdConfig/cli/cStylePromptInJStyle` (xs:boolean) [false]
If set to true then the \m and \M will be expanded just as in C- and I-style

`/confdConfig/cli/promptHostnameDelimiter` (xs:string) [.]
When the \h token is used in a prompt the first part of the hostname up until the first occurrence of the promptHostnameDelimiter is used.

`/confdConfig/cli/asyncPromptRefresh` (xs:boolean) [true]
asyncPromptRefresh is either "true" or "false". If set to "true" the CLI prompt will be refreshed when asynchronous tasks prints messages in the CLI, such as messages from other users.

`/confdConfig/cli/showLogDirectory` (xs:string)
Location where the 'show log' command looks for log files.

`/confdConfig/cli/modeInfoInAudit` (true|false|path) [false]
modeInfoInAudit is either "true", "false" or "path". If "true", then all commands will be prefixed with major and minor mode name when logged as audit messages. This means that it is possible to differentiate between commands with the same name in different modes. Major mode is "operational" or "configure" and minor mode is "top" in J-style and the name of the submode in C- and I-mode. On the top-level in C- and I-mode it is also "top". If set to "path" the major mode will be followed by the full command path to the submode.

`/confdConfig/cli/auditLogMode` (all|allowed|denied) [all]
auditLogMode is either "all", "allowed", or "denied". If "all", then all commands that the user tries to execute will be logged in the audit trail log. If "allowed", only allowed commands will be logged, ie commands that are actually run by the user. If "denied", only commands that the user were not allowed to execute will be logged, prefixed with "denied".

`/confdConfig/cli/style` (j|c|i) [j]
style is either "j", "c", or "i". If "j", then the CLI will be presented as a Juniper style CLI. If "c" then the CLI will appear as Cisco XR style, and if "i" then a Cisco IOS style CLI will be rendered.

`/confdConfig/cli/idleTimeout` (xs:duration) [PT30M]
Maximum idle time before terminating a CLI session. Default is PT30M, ie 30 minutes. PT0M means no timeout. Will be silently capped to 49 days 17 hours

`/confdConfig/cli/promptSessionsCLI` (xs:boolean) [false]
promptSessionsCLI is either "true" or "false". If set to "true" then only the current CLI sessions will be displayed when the user tries to start a new CLI session and the maximum number of sessions has been reached. Note that MAAPI sessions with their context set to "cli" would be regarded as CLI sessions and would be listed as such.

`/confdConfig/cli/disableIdleTimeoutOnCmd` (xs:boolean) [true]
disableIdleTimeoutOnCmd is either "true" or "false". If set to "false" then the idle timeout will trigger even when a command is running in the CLI. If set to "true" the idle timeout will only trigger if the user is idling at the CLI prompt.

`/confdConfig/cli/commandTimeout` (xs:duration|infinity) [infinity]
Global command timeout. Terminate command unless the command has completed within the timeout. It is generally a bad idea to use this feature since it may have undesirable effects in a loaded system where normal commands take longer to complete than usual.

This timeout can be overridden by a command specific timeout specified in the confd.cli file.

`/confdConfig/cli/commitRetryTimeout` (xs:duration|infinity) [PT0S]
Commit timeout in the CLI. This timeout controls for how long the commit operation will attempt to complete the operation when some other entity is locking the database, e.g. some other commit is in progress or some managed object is locking the database.

There is a similiar configuration parameter, `/confdConfig/commitRetry/Timeout`, which sets a timeout for all ConfD transactions, not just for CLI transactions.

`/confdConfig/cli/timezone` (utc|local) [local]
Used to specify which timezone should be used when displaying the time in the CLI. If "local" is specified then the timezone that is configured on the device will be used.

`/confdConfig/cli/utcOffset` (xs:integer) [0]
If the timezone is set to UTC this can be set to specify the UTC offset measured in minutes.

`/confdConfig/cli/timestamp - container element`
Default value for the timstamps in the CLI. The user can always enable or disable the display of timestamps, this only controls the initial session value.

`/confdConfig/cli/timstamp/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true" the CLI will print a timestamp before the output of each command.

`/confdConfig/cli/spaceCompletion - container element`
Default value for space completion in the CLI. The user can always enable or disable completion on space, this only controls the initial session value.

`/confdConfig/cli/spaceCompletion/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true" command and argument completion will be performed when <space> is entered.

`/confdConfig/cli/autoWizard - container element`
Default value for autowizard in the CLI. The user can always enable or disable the auto wizard in each session, this controls the initial session value.

`/confdConfig/cli/autoWizard/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true" the CLI will prompt the user for required attributes when a new identifier is created.

`/confdConfig/cli/ssh/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true" ConfD will will run the builtin SSH daemon and run the CLI.

`/confdConfig/cli/ssh/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the ConfD CLI should listen to for SSH sessions. 0.0.0.0 means that it listens to the port (/confdConfig/cli/ssh/port) for all IPv4 addresses on the machine.

`/confdConfig/cli/ssh/port` (confd:inetPortNumber) [2024]
The port number for CLI SSH

`/confdConfig/cli/ssh/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the CLI also listens to for SSH connections. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/cli/ssh/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
```

```
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/cli/ssh/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the CLI for SSH connections.

`/confdConfig/cli/ssh/banner` (xs:string) []
banner is a string that will be presented to the client before authenticating when logging in to the CLI via the built-in SSH server.

`/confdConfig/cli/ssh/bannerFile` (xs:string)
Name of file whose contents will be presented to the client before authenticating when logging in to the CLI via the built-in SSH server. If /confdConfig/cli/ssh/banner is non-empty, its value will be shown before the contents of this file. Default is empty.

`/confdConfig/cli/showCommitProgress` (xs:boolean) [false]
showCommitProgress can be either "true" or "false". If set to "true" then the commit operation in the CLI will provide some progress information when the output is piped to the 'details' target.

`/confdConfig/cli/commitActivityClock` (xs:boolean) [false]
commitActivityClock can be either "true" or "false". If set to "true" then a |/-\| style animation will be displayed if the commit operation takes more than 200 ms to complete.

`/confdConfig/cli/commitMessage` (xs:boolean) [true]
commitMessage is either "true" or "false". If set to "true" then a message will be displayed in the CLI whenever a commit operation is performed in the system. This is always disabled in I-style, and in transactionless mode.

`/confdConfig/cli/commitMessageFormat` (xs:string) [System message at $(time)...\nCommit performed by $(user) via $(proto) using $(ctx).\n]
commitMessageFormat controls how commit messages are displayed in the CLI. The format string may contain the variables $(user), $(time), $(ctx), $(date), $(time12), $(time12ampm), $(time12hm), $(host), $(hostname), and $(proto).

`/confdConfig/cli/suppressCommitMessages - container element`
Suppress commit messages from certain contexts.

`/confdConfig/cli/suppressCommitMessages/context` (xs:string) []
Suppress commit messages from a certain context. The value of suppressCommitMessages should be the name of a context. For example "system".

`/confdConfig/cli/jStatusFormat` (xs:string) [[$(status)][$(time)]\n]
jStatusFormat controls which status message is displayed after executing a CLI command in the J-style CLI. The format string may contain the variables $(status), $(time), $(date), $(time12), $(time12ampm), $(time12hm), $(host), $(hostname), and $(proto).

`/confdConfig/cli/forcedExitFormat` (xs:string) [You are forced out of configure mode by $(user).\n]
forceExitFormat controls which message to display when a user is forced out of configure mode by another user. The format string may contain the variables $(user), $(time), $(date), $(time12), $(time12ampm), $(time12hm), $(host) and $(hostname).

`/confdConfig/cli/showSubsystemMessages` (xs:boolean) [true]
showSubsystemMessages is either "true" or "false". If "true" the CLI will display a system message whenever a connected daemon is started or stopped.

`/confdConfig/cli/ignoreSubsystemFailures` (xs:boolean) [false]
ignoreSubsystemFailures is either "true" or "false". If "true" the CLI will make a best effort to display data even if a data provider is unavailable.

`/confdConfig/cli/showEmptyContainers` (xs:boolean) [false]
showEmptyContainers is either "true" or "false". If "true" the CLI will display empty container nodes when displaying the configuration. If "false" then empty static containers will not be shown.

`/confdConfig/cli/showTags` (xs:boolean) [true]
showTags is either "true" or "false". If "true" the CLI will display configuration tags if they are present. If set to "false" then the tags will not be displayed by default.

`/confdConfig/cli/showAnnotations` (xs:boolean) [true]
showAnnotations is either "true" or "false". If "true" the CLI will display configuration annotations if they are present. If set to "false" then the annotations will not be displayed by default.

`/confdConfig/cli/orderedShowConfig` (xs:boolean) [true]
orderedShowConfig is either "true" or "false". If "true" then the commands displayed when running the "show configuration" command in C-mode will take leafrefs and cli-diff-dependency into account.

`/confdConfig/cli/suppressFastShow` (xs:boolean) [false]
suppressFastShow is either "true" or "false". If "true" then the fast show optimization will be suppressed in the C-style CLI. The fast show optimization is somewhat experimental and may break certain operations.

`/confdConfig/cli/leafPrompting` (xs:boolean) [true]
leafPrompting is either "true" or "false". If "true" the CLI will prompt the user for leaf values if they are not provided on the command line. If "false" then an error message will be displayed if the user does not provide a value for a leaf.

`/confdConfig/cli/jExtendedShow` (xs:boolean) [true]
jExtendedShow is either "true" or "false". If set to "true" then the J-style CLI will have auto-rendered show commands in the same style as the C and I-style CLIs. The 'show status' command can still be used for viewing config="false" data.

`/confdConfig/cli/jShowCR` (xs:boolean) [false]
jShowCR is either "true" or "false". If set to "true" then the J-style CLI will show >cr< in the completion list whenever it is legal to press cr.

`/confdConfig/cli/showPipe` (xs:boolean) [true]
showPipe is either "true" or "false". If set to "true" in operational mode the completion list will contain | if it is legal to enter |. In J-style the jShowCR must also be set to enable this. If disablePipe is set, it will override the setting of showPipe and imply the same behavior as if showPipe is "false".

`/confdConfig/cli/showPipeConfig` (xs:boolean) [false]
showPipeConfig is either "true" or "false". If set to "true" in configure mode the completion list will contain | if it is legal to enter |. In J-style the jShowCR must also be set to enable this. If disablePipeConfig is set, it will override the setting of showPipeConfig and imply the same behavior as if showPipeConfig is "false".

`/confdConfig/cli/disablePipe` (xs:boolean) [false]
disablePipe is either "true" or "false". If set to "true" then the pipe commands are disabled in operational mode.

`/confdConfig/cli/disablePipeConfig` (xs:boolean) [false]
disablePipeConfig is either "true" or "false". If set to "true" then the pipe commands are disabled in configure mode.

`/confdConfig/cli/pipeHelpMode` (pipeHelpModeType) [auto]
If showPipe is set to true, then pipeHelpMode determines how the pipe option will be displayed to the user. If set to "auto", then the description text "Output modifiers" will only be displayed if there are any other options with help texts, otherwise it will not be shown. If set to "always" then the help text will always be displayed, if set to "never" then it will never be displayed.

`/confdConfig/cli/jAllowDeleteAll` (xs:boolean) [true]
jAllowDeleteAll is either "true" or "false". If set to "true" then the J-style CLI will show the command "delete" without arguments, if set to "false" then an argument is required.

`/confdConfig/cli/cSilentNo` (xs:boolean) [true]
Silently ignore deletes of non-existing instances.

`/confdConfig/cli/cModeExitFormat` (xs:string) [!]
cModeExitFormat is the string used in the CLI when displaying the running configuration to indicate exit from a submode.

`/confdConfig/cli/cRestrictiveNo` (xs:boolean) [false]
If a leaf value is given when an optional leaf is deleted, the given value is normally ignored and the node is deleted regardless of the value. When cRestrictiveNo is set to "true", the given value is required to be equal to the old value in order to the delete operation to be allowed. For example the Cisco style command "no interface eth0 mtu 1500" will only succeed if the mtu has the value 1500.

`/confdConfig/cli/cExtendedCmdSearch` (xs:boolean) [true]
Extend the available submode commands to all commands in parent (and grand-parent) modes. These commands are not visible during completion but will be executed if entered. If set to "false" then only commands for entering other submodes are available in parent and grand-parent modes, if set to "true" all commands in parent and grand-parent modes are available.

`/confdConfig/cli/cSuppressCmdSearch` (xs:boolean) [false]
Prevent non-local commands from being executed. This negates the effect of cExtendedCmdSearch above. It is recommended to also set cModeExitFormat to "exit" when this option is set to true.

`/confdConfig/cli/enterSubmodeOnLeaf` (xs:boolean) [true]
enterSubmodeOnLeaf is either "true" or "false". If set to "true" (the default) then setting a leaf in a submode from a parent mode results in entering the submode after the command has completed. If set to "false" then an explicit command for entering the submode is needed. For example, if running the command

```
interface FastEthernet 1/1/1 mtu 1400
```

from the top level in config mode. If enterSubmodeOnLeaf is true the CLI will end up in the "interface FastEthernet 1/1/1" submode after the command execution. If set to "false" then the CLI will remain at the top level. To enter the submode when set to "false" the command

```
interface FastEthernet 1/1/1
```

is needed. Applied to the C- and I- style CLI.

`/confdConfig/cli/noFollowIncompleteCommand` (xs:boolean) [false]
noFollowIncompleteCommand is either "true" or "false". If set to "true" then the 'no' command will take incomplete-command declarations into account. If set to "false" it will not.

`/confdConfig/cli/jShowUnset` (xs:boolean) [false]
jShowUnset is either "true" or "false". If set to "true" then the J-style CLI will show unset leaves with the value of jShowUnsetText when doing "show configuration".

`/confdConfig/cli/jShowUnsetText` (xs:string) [UNSET]
jShowUnsetText is the text printed for unset values if jShowUnset has been set to true.

`/confdConfig/cli/jShowTableRecursive` (xs:boolean) [false]
jShowTableRecursive is either "true" or "false". If "true" the J-style CLI will attempt to display the result of the command "show table" as a table even when a list is not directly specified. If set to "false" then a table will only be produced if a list node is specified as argument to "show table".

`/confdConfig/cli/cPrivate` (xs:boolean) [false]
cPrivate is either "true" or "false". If set to "true" then the term "private" will be used in place of "terminal" for denoting the private/terminal configuration mode. When set to "false" the term "terminal" will be used instead.

`/confdConfig/cli/cTab` (xs:boolean) [false]
cTab is either "true" or "false". If "true" the Cisco style CLI will not display any help text when the user enters TAB. If "false" then help text will be shown when entering TAB, similarly to the Juniper-style CLI.

`/confdConfig/cli/cTabInfo` (xs:boolean) [false]
cTabInfo is either "true" or "false". If "false" the Cisco style CLI will not display any info text when the user enters TAB. If "true" then info text will be shown when entering TAB, similarly to the Juniper-style CLI.

`/confdConfig/cli/tabExtend` (xs:boolean) [false]
tabExtend is either "true" or "false". If "true" the CLI will extend the current token to the next longer alternative.

`/confdConfig/cli/cHelp` (xs:boolean) [true]
cHelp is either "true" or "false". If "true" the Cisco style CLI will not display any desc text when the user enters '?'. If "false" then desc text will be shown when entering '?', similarly to the Juniper-style CLI.

`/confdConfig/cli/restrictedFileAccess` (xs:boolean) [false]
restrictedFileAccess is either "true" or "false". If "true" then a CLI user will not be able to access files and directories outside the home directory tree.

`/confdConfig/cli/hideDotFiles` (xs:boolean) [false]
hideDotFile is either "true" or "false". If "true" then files starting with a '.' will not be visible in the CLI..

`/confdConfig/cli/restrictedFileRegexp` (xs:string) []
restrictedFileRegexp is either an empty string or an regular expression (AWK style). If not empty then all files and directories created or accessed must match the regular expression. This can be used to ensure that certain symbols does not occur in created file names.

`/confdConfig/cli/mapActions` (both|config|oper) [both]
mapActions is either "both", "config", or "oper". If "both", then actions are available both in operational mode and in configure mode. If "oper" then they are only available in operational mode, and if "config" then they are only available in configure mode.

`/confdConfig/cli/modeNameStyle` (full|short|two) [short]
modeNameStyle is either "short", "two", or "full". If "short", then the mode name of submodes in the Cisco style CLIs will be constructed from the last element in the path and the instance key. If set to "two" then the two last modes will be used for the mode name. If set to "full" then all components in the path will be used in the mode name.

`/confdConfig/cli/modeInfoInAAA` (true|false|path) [false]
modeInfoInAAA is either "true", "false" or "path", If "true", then all commands will be prefixed with major and minor mode name when processed by the AAA-rules. This means that it is possible to differentiate between commands with the same name in different modes. Major mode is "operational"

or "configure" and minor mode is "top" in J-style and the name of the submode in C- and I-mode. On the top-level in C- and I-mode it is also "top". If set to "path" and if the command operation is "read" the major mode will be followed by the path to the submode which will be followed by the command. If set to "path" and if the command operation is "execute" the major mode will instead be followed by the command and the path to the submode will be prepended to any path arguments of the command.

`/confdConfig/cli/cmdAAAForAutowizard` (true|false) [false]
cmdAAAForAutowizard si either "true" or "false". If set to "true" then the CLI will generate synthetic commands, and perform AAA command rule checks for, for all paths and values requested by the autowizard functionality.

`/confdConfig/cli/quoteStyle` (quote|baskslash) [backslash]
quoteStyle is either "quote" or "backslash". If set to "quote" then the quote characters will be used on the CLI command line for quoting strings with troublesome characters. If set to "backslash" then a backslash will be used. For example:

Using quote:

```
io(config)# description "description with spaces"
```

Using backslash:

```
io(config)# description description\ with\ spaces
```

`/confdConfig/cli/laxBarQuoting` (xs:boolean) [false]
laxBarQuoting is either "true" or "false". If set to "true" then | and ; are only quoted if they appear by them selves. A consequence of this is that the user must have whitespace on both sides of | and ; on the command line when these characters are used as pipe (|) or concatenator (;).

`/confdConfig/cli/expandAliasOnCompletion` (xs:boolean) [true]
expandAliasOnCompletion is either true or false. If set to true then aliases will be expanded before invoking the completion code.

`/confdConfig/cli/expandAliasEscape` (xs:boolean) [false]
expandAliasEscape is either false or a character. If set to a character then expanding an alias can be prevented by putting the character in front of the alias.

`/confdConfig/cli/allowParenQuotes` (xs:boolean) [false]
allowParenQuotes is either "true" or "false". If set to "true" then parentheses are treated as quotes, ie the string (xx yy) will be equivalent to "xx yy" and xx\ yy on the CLI command line.

`/confdConfig/cli/execNavigationCmds` (xs:boolean) [false]
execNavigationCmds is either "true" or "false". If set to "true" then it is possible to enter a submode also in exec mode in C- and I-style CLI.

`/confdConfig/cli/exitConfigModeOnCtrlC` (xs:boolean) [true]
exitConfigModeOnCtrlC is either "true" or "false". If set to "false" the user will not be thrown out of config mode when ctrl-c is pressed on an empty command line.

`/confdConfig/cli/allowOvewriteOnCopy` (xs:boolean) [false]
allowOverwriteOnCopy is either "true" or "false". If set to "true" then the copy command in the CLI will overwrite the target if it exists. If set to "false" then an error will be displayed if the target exists.

`/confdConfig/cli/inheritPaginate` (xs:boolean) [false]
inheritPaginate is either "true" or "false". If set to "true" then the paginate setting of a pipe command will be determined by the paginate setting of the main command. If set to "false", then the output from a pipe command will not be paginated unless pagination for that pipe command has been overridden in a clispec file.

`/confdConfig/aaa - container element`
The login procedure to ConfD is fully described in the ConfD User's Guide.

`/confdConfig/aaa/sshLoginGraceTime` (xs:duration) [PT10M]
ConfD closes ssh connections after this time if the client has not successfully authenticated itself by then. If the value is PT0S, there is no time limit for client authentication.

This is a global value for all ssh servers in ConfD.

Modification of this value will only affect ssh connections that are established after the modification has been done.

`/confdConfig/aaa/sshMaxAuthTries` (xs:unsignedInt|unbounded) [unbounded]
ConfD closes ssh connections when the client has made this number of unsuccessful authentication attempts.

This is a global value for all ssh servers in ConfD.

Modification of this value will only affect ssh connections that are established after the modification has been done.

`/confdConfig/aaa/sshPubkeyAuthentication` (none|local|system) [system]
Controls how the ConfD SSH daemon locates the user keys for public key authentication.

If set to "none", public key authentication is disabled.

If set to "local", and the user exists in `/aaa/authentication/users`, the keys in the user's 'ssh_keydir' directory are used.

If set to "system", the user is first looked up in `/aaa/authentication/users`, but only if `/confdConfig/aaa/localAuthentication/enabled` is set to "true" - if `localAuthentication` is disabled, or the user does not exist in `/aaa/authentication/users`, but the user does exist in the OS password database, the keys in the user's $HOME/.ssh directory are used.

`/confdConfig/aaa/sshServerKeyDir` (xs:string)
sshServerKeyDir is the directory file path where the keys used by the ConfD SSH daemon are found. This parameter must be set if SSH is enabled for NETCONF or the CLI. If SSH is enabled, the server keys used by ConfD are of the same format as the server keys used by openssh, i.e. the same format as generated by 'ssh-keygen'.

Note

Only DSA-type keys can be used with the ConfD SSH daemon, as generated by 'ssh-keygen' with the '-t dsa' switch.

The key must be stored with an empty passphrase.

`/confdConfig/aaa/defaultGroup` (xs:string)
If the group of a user cannot be found in the AAA sub-system, a logged in user will end up as a member of the default group (if specified). If a user logs in and the group membership cannot be established, the user will have zero access rights.

`/confdConfig/aaa/authOrder` (xs:string)
By default the AAA system will try to authenticate a user in the following order. (1) localAuthentication i.e. the user is found inside /aaa/authentication/users. (2) pam - i.e PAM authentication - if enabled - is tried. (3) externalAuthentication i.e. an external program is invoked to authenticate the user.

The default is thus:

"localAuthentication pam externalAuthentication"

To change the order - change this string. For example in order to always try pam authentication before local auth set it to: "pam localAuthentication"

`/confdConfig/aaa/localAuthentication/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", local authentication is used, i.e. the user data kept in the aaa namespace is used to authenticate users. If "false" some other authentication mechanism such as PAM or external authentication must be used.

`/confdConfig/aaa/pam - container element`
If PAM is to be used for login the ConfD daemon typically must run as root.

`/confdConfig/aaa/pam/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", ConfD uses PAM for authentication.

`/confdConfig/aaa/pam/service` (xs:string) [common-auth]
The PAM service to be used for the authentication. This can be any service we have installed in the `/etc/pam.d` directory. Different unices have different services installed under `/etc/pam.d`, and some use a file `/etc/pam.conf` instead - choose a service which makes sense or create a new one.

`/confdConfig/aaa/pam/timeout` (xs:duration) [PT10S]
The maximum time that authentication will wait for a reply from PAM. If the timeout is reached, the PAM authentication will fail, but authentication attempts may still be done with other mechanisms as configured for `/confdConfig/aaa/authOrder`. Default is PT10S, i.e. 10 seconds.

`/confdConfig/aaa/externalAuthentication/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", external authentication is used.

`/confdConfig/aaa/externalAuthentication/executable` (xs:string)
If we enable external authentication, an executable on the local host can be launched to authenticate a user. The executable will receive the username and the cleartext password on its standard input. The format is "[${USER};${PASS};]\n". For example if user is "bob" and password is "secret", the executable will receive the string "[bob;secret;]" followed by a newline on its standard input. The program must parse this line.

The task of the external program, which for example could be a RADIUS client, is to authenticate the user and also provide the user to groups mapping. Refer to the External authentication section of the AAA chapter in the User Guide for the details of how the program should report the result back to ConfD.

`/confdConfig/aaa/externalAuthentication/useBase64` (xs:boolean) [false]
When set to "true", ${USER} and ${PASS} in the data passed to the executable will be base64-encoded, allowing e.g. for the password to contain ';' characters. For example if user is "bob" and password is "secret", the executable will receive the string "[Ym9i;c2VjcmV0;]" followed by a newline.

`/confdConfig/aaa/externalAuthentication/includeExtra` (xs:boolean) [false]
When set to "true", additional information items will be provided to the executable: source IP address and port, context, and protocol. I.e. the complete format will be "[${USER};${PASS};${IP};${PORT};${CONTEXT};${PROTO};]\n". Example: "[bob;secret;192.168.1.1;12345;cli;ssh;]\n".

`/confdConfig/aaa/authenticationCallback/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", ConfD will invoke an application callback when authentication has succeeded or failed. The callback may reject an otherwise successful authentication. If the callback has not been registered, all authentication attempts will fail. See confd_lib_dp(3) for the callback details.

`/confdConfig/aaa/authorization/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "false", all authorization checks are turned off similar to the -noaaa flag in confd_cli.

`/confdConfig/aaa/authorization/callback/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", ConfD will invoke application callbacks for authorization. If the callbacks have not been registered, all authorization checks will be rejected. See confd_lib_dp(3) for the callback details.

`/confdConfig/aaa/aaaBridge - container element`
aaaBridge specifies if the aaa_bridge, as described in confd_aaa_bridge(1), will be used to access external AAA info.

`/confdConfig/aaa/aaaBridge/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the confd_aaa_bridge program is automatically started by ConfD

`/confdConfig/aaa/aaaBridge/file` (xs:string)
file specifies the location of the AAA data file needed by the confd_aaa_bridge program.

`/confdConfig/aaa/namespace` (xs:string) [`http://tail-f.com/ns/aaa/1.1`]
If we want to move the AAA data into another userdefined namespace, we indicate that here.

`/confdConfig/aaa/prefix` (xs:string) [/]
If we want to move the AAA data into another userdefined namespace, we indicate the prefix path in that namespace where the ConfD AAA namespace has been mounted.

This feature has been deprecated, it is easier to achieve this through a symlink.

`/confdConfig/netconf - container element`
This section defines settings which decide how the NETCONF agent should behave, with respect to NETCONF and SSH.

`/confdConfig/netconf/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the NETCONF agent is started.

`/confdConfig/netconf/versions - container element`
The list of NETCONF versions that the NETCONF server will understand and advertise.

`/confdConfig/netconf/versions/v1.0` (xs:boolean) [true]
Setting the value to true will enable NETCONF version 1.0, as defined in RFC 4741.

`/confdConfig/netconf/versions/v1.1` (xs:boolean) [true]
Setting the value to true will enable NETCONF version 1.1, as defined in RFC 6241.

`/confdConfig/netconf/transport - container element`
Settings deciding which transport services the NETCONF agent should listen to, e.g. TCP and SSH.

`/confdConfig/netconf/transport/ssh - container element`
Settings deciding how the NETCONF SSH transport service should behave.

`/confdConfig/netconf/transport/ssh/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the NETCONF agent uses SSH as a transport service.

`/confdConfig/netconf/transport/ssh/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the ConfD NETCONF agent should listen to. 0.0.0.0 means that it listens to the port (/confdConfig/netconf/transport/ssh/port) for all IPv4 addresses on the machine.

`/confdConfig/netconf/transport/ssh/port` (confd:inetPortNumber) [2022]
port is a valid port number to be used in combination with /confdConfig/netconf/transport/ssh/ip. Note that the standard port for NETCONF over SSH is 830.

`/confdConfig/netconf/transport/ssh/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the NETCONF agent also listens to for SSH connections. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/netconf/transport/ssh/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/netconf/transport/ssh/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the NETCONF server for SSH connections.

`/confdConfig/netconf/transport/tcp - container element`
NETCONF over TCP is not standardized, but it can be useful during development in order to use e.g. netcat for scripting. It is also useful if we want to use our own proprietary transport. In that case we setup the NETCONF agent to listen to localhost and then proxy it from our transport service module.

`/confdConfig/netconf/transport/tcp/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the NETCONF agent uses clear text TCP as a transport service.

`/confdConfig/netconf/transport/tcp/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the ConfD NETCONF agent should listen to. 0.0.0.0 means that it listens to the port (/confdConfig/netconf/transport/tcp/port) for all IPv4 addresses on the machine.

`/confdConfig/netconf/transport/tcp/port` (confd:inetPortNumber) [2023]
port is a valid port number to be used in combination with /confdConfig/netconf/transport/tcp/ip.

`/confdConfig/netconf/transport/tcp/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the NETCONF agent also listens to for TCP connections. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/netconf/transport/tcp/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/netconf/transport/tcp/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the NETCONF server for TCP connections.

`/confdConfig/netconf/capabilities - container element`
Decide which NETCONF capabilities to enable here.

`/confdConfig/netconf/capabilities/startup/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the startup NETCONF capability is enabled. Enable only if /confdConfig/datastores/startup is enabled.

`/confdConfig/netconf/capabilities/candidate/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the candidate NETCONF capability is enabled. Enable only if /confdConfig/datastores/candidate is enabled.

`/confdConfig/netconf/capabilities/confirmed-commit/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the confirmed-commit NETCONF capability is enabled.

`/confdConfig/netconf/capabilities/writable-running/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the writable-running NETCONF capability is enabled. Enable only if /confdConfig/datastores/running/access is read-write.

`/confdConfig/netconf/capabilities/rollback-on-error/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the rollback-on-error NETCONF capability is enabled.

`/confdConfig/netconf/capabilities/validate/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If enabled "true", the validate NETCONF capability is enabled.

`/confdConfig/netconf/capabilities/validate/test-only` (xs:boolean) [false]
DEPRECATED - this feature is available in NETCONF 1.1.

If test-only "true", the NETCONF agent allows 'test-only' as a valid value for the '<test-option>' parameter in '<edit-config>'.

`/confdConfig/netconf/capabilities/url - container element`
Turn on the URL capability options we want to support.

`/confdConfig/netconf/capabilities/url/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the url NETCONF capability is enabled.

`/confdConfig/netconf/capabilities/url/file - container element`
Decide how the url file support should behave.

`/confdConfig/netconf/capabilities/url/file/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the url file scheme is enabled.

`/confdConfig/netconf/capabilities/url/file/rootDir` (xs:string)
rootDir is a directory path on disk where ConfD will store the result from an NETCONF operation using the url capability. This parameter must be set if the file url scheme is enabled.

`/confdConfig/netconf/capabilities/url/ftp - container element`
Decide how the url ftp scheme should behave.

`/confdConfig/netconf/capabilities/url/ftp/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the url ftp scheme is enabled.

`/confdConfig/netconf/capabilities/url/sftp - container element`
Decide how the url sftp scheme should behave.

`/confdConfig/netconf/capabilities/url/sftp/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the url sftp scheme is enabled.

`/confdConfig/netconf/capabilities/xpath/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the xpath capability is enabled.

`/confdConfig/netconf/capabilities/notification/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the notification capability, defined in RFC 5277, is enabled.

`/confdConfig/netconf/capabilities/notification/interleave/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the interleave capability, defined in RFC 5277, is enabled. With this capability enabled, the NETCONF agent will process RPCs while sending notifications.

`/confdConfig/netconf/capabilities/partial-lock/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the partial-lock capability defined in RFC 5717 is enabled.

`/confdConfig/netconf/capabilities/actions/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/actions/1.0` capability is enabled.

>`/confdConfig/netconf/capabilities/transactions/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/transactions/1.0` capability is enabled.

`/confdConfig/netconf/capabilities/ietf-with-defaults/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the with-defaults capability, defined in RFC 6243, is enabled.

The NETCONF server will advertise its 'basic-mode' and 'also-supported' modes depending on the parameter `/confdConfig/defaultHandlingMode`.

`/confdConfig/netconf/capabilities/with-defaults/enabled` (xs:boolean) [false]
DEPRECATED - use ietf-with-defaults instead.

enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/with-defaults/1.0` capability is enabled.

`/confdConfig/netconf/capabilities/forward/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/forward/1.0` capability is enabled.

`/confdConfig/netconf/capabilities/tailf-commit/enabled` (xs:boolean) [false]
DEPRECATED - this feature is available in NETCONF 1.1.

enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/commit/1.0` capability is enabled.

`/confdConfig/netconf/capabilities/query/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/query` capability is enabled.

`/confdConfig/netconf/capabilities/inactive/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the `http://tail-f.com/ns/netconf/inactive/1.0` capability is enabled.

`/confdConfig/netconf/capabilities/capability` (xs:anyURI)
capability is a parameter can be given multiple times. It specifies a URI string which the NETCONF agent will report as a capability in the hello message sent to the client.

`/confdConfig/netconf/maxBatchProcesses`(xs:unsignedInt|unbounded) [unbounded]"
Controls how many concurrent NETCONF batch processes there can be at any time. A batch process can be started by the agent if a new NETCONF operation is implemented as a batch operation. See the NETCONF chapter in the ConfD User's Guide for details.

`/confdConfig/netconf/extendedSessions` (xs:boolean) [false]
If extendedSessions are enabled, all ConfD sessions can be terminated using <kill-session>, i.e. not only can other NETCONF session be terminated, but also CLI sessions, Webui sessions etc. If such a session holds a lock, it's session id will be returned in the <lock-denied>, instead of "0".

Strictly speaking, this extension is not covered by the NETCONF specification; therefore it's false by default.

`/confdConfig/netconf/sendDefaults` (xs:boolean) [false]
DEPRECATED - use `/confdConfig/defaultHandlingMode` instead to control this behavior consistently for all northbound interfaces.

If sendDefaults is true, default values will be included in the replies to <get>, <get-config>, and <copy-config>. If sendDefaults is false, default values will not be included by default.

If /confdConfig/netconf/capabilities/with-defaults is enabled, this behavior can be controlled by the NETCONF client.

`/confdConfig/netconf/rpcErrors` (close|inline) [close]
If rpcErrors is "inline", and an error occurs during the processing of a <get> or <get-config> request when ConfD tries to fetch some data from a data provider, ConfD will generate an rpc-error element in the faulty element, and continue to process the next element.

If an error occurs and rpcErrors is "close", the NETCONF transport is closed by ConfD.

`/confdConfig/netconf/idleTimeout` (xs:duration) [PT0S]
Maximum idle time before terminating a NETCONF session. If the session is waiting for notifications, or has a pending confirmed commit, the idle timeout is not used. The default value is 0, which means no timeout.

`/confdConfig/netconf/proxyIdleTimeout` (xs:duration) [PT0S]
Maximum idle time before terminating a NETCONF session which is acting as a proxy. The default is 0, which means no timeout.

`/confdConfig/proxyForwarding - container element`
This section defines settings which affect the behavior of Proxy Forwarding.

`/confdConfig/proxyForwarding/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", proxy forwarding is enabled.

`/confdConfig/proxyForwarding/autoLogin` (xs:boolean) [false]
autoLogin is either "true" or "false". If "true", ConfD will try to login to the target system with the current sessions credentials, if it has access to them. In order for ConfD to get access to the session credentials, the builtin SSH daemon must be used.

`/confdConfig/proxyForwarding/proxy - container element`
Parameters for a single proxy.

`/confdConfig/proxyForwarding/proxy/target - container element`
The name of the proxy target. It is used as a unique identifier of the proxy target. This is the target name that users give when they want to connect to the target.

The name is included in the proxy events (see confd_lib_events(3)) generated by ConfD.

`/confdConfig/proxyForwarding/proxy/address (confd:inetAddress)`
The IP address of the proxy target system.

`/confdConfig/proxyForwarding/proxy/netconf - container element`
If present, the target is available for NETCONF proxy forwarding.

`/confdConfig/proxyForwarding/proxy/netconf/ssh - container element`
If present, the proxy connections between ConfD and the proxy NETCONF target will be over SSH.

`/confdConfig/proxyForwarding/proxy/netconf/ssh/port (confd:inetPortNumber) [2022]`
The port where the proxy target listens for NETCONF SSH connections.

`/confdConfig/proxyForwarding/proxy/netconf/tcp - container element`
If present, the proxy connections between ConfD and the proxy NETCONF target will be over TCP.

`/confdConfig/proxyForwarding/proxy/netconf/tcp/port (confd:inetPortNumber) [2023]`
The port where the proxy target listens for NETCONF TCP connections.

`/confdConfig/proxyForwarding/proxy/cli - container element`
If present, the target is available for CLI proxy forwarding.

`/confdConfig/proxyForwarding/proxy/cli/ssh - container element`
If present, the proxy connections between ConfD and the proxy CLI target will be over SSH.

`/confdConfig/proxyForwarding/proxy/cli/ssh/port (confd:inetPortNumber) [22]`
The port where the proxy target listens for CLI SSH connections.

`/confdConfig/snmpAgent - container element`
This section defines settings which affect the behavior of the SNMP agent.

`/confdConfig/snmpAgent/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the SNMP agent is enabled.

`/confdConfig/snmpAgent/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the ConfD SNMP agent should listen to. 0.0.0.0 means that it listens to the port (/confdConfig/snmpAgent/port) for all IPv4 addresses on the machine.

`/confdConfig/snmpAgent/port` (confd:inetPortNumber) [161]
port is a valid port number to be used in combination with /confdConfig/snmpAgent/ip.

`/confdConfig/snmpAgent/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the SNMP agent also listens to. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/snmpAgent/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:
i

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/snmpAgent/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the SNMP agent.

`/confdConfig/snmpAgent/mibs - container element`
This section defines a list of MIBs that should be loaded into the SNMP agent.

`/confdConfig/snmpAgent/mibs/file` (xs:string)
file is the location of a MIB file that should be loaded into the SNMP agent. For example:

```
<file>./TAIL-F-TEST-MIB.bin</file>
```

The MIB file must be in binary format (.bin) produced with the confdc compiler. For loading of a built-in MIB no path must be given. Example:

```
<file>SNMP-USER-BASED-SM-MIB.bin</file>
```

See the ConfD User's Guide for more information about loading MIBs into the SNMP agent.

`/confdConfig/snmpAgent/mibs/fromLoadPath` (xs:boolean) [false]
If true, any ".bin" file found in the /confdConfig/loadPath is loaded at startup. Bult-in MIBs must still be listed explicitly using the "file" element.

`/confdConfig/snmpAgent/temporaryStorageTime` (xs:unsignedInt) [300]
The time, in seconds, that the agent keeps temporary table entries before deleting them. A table entry is temporary if its RowStatus column is 'notReady' or 'notInService'.

`/confdConfig/snmpAgent/sessionIgnorePort` (xs:boolean) [false]
If true, the SNMP Agent will consider requests originating from one and the same IP Address, and using the same security name, as related, regardless of source port. Per default, the SNMP Agent will consider requests originating from one and the same IP Address and port, and using the same security name, as related. Related requests are handled in the same user session. This is absolutely necessary for achieving good performance when processing consecutive get-next requests, as during SNMP walks.

`/confdConfig/snmpAgent/snmpVersions - container element`
This section defines the list of SNMP versions that the SNMP agent should understand.

`/confdConfig/snmpAgent/snmpVersions/v1` (xs:boolean) [true]
Setting the value to true will enable SNMP v1 in the SNMP agent.

`/confdConfig/snmpAgent/snmpVersions/v2c` (xs:boolean) [true]
Setting the value to true will enable SNMP v2c in the SNMP agent.

`/confdConfig/snmpAgent/snmpVersions/v3` (xs:boolean) [true]
Setting the value to true will enable SNMP v3 in the SNMP agent.

`/confdConfig/snmpAgent/snmpEngine - container element`
This section defines properties from the SNMP-FRAMEWORK-MIB (RFC3411) for the SNMP agent.

`/confdConfig/snmpAgent/snmpEngine/snmpEngineID` (confd:hexlist) [
The name of the SNMP engine. snmpEngineID is defined in the SNMP-FRAMEWORK-MIB (RFC3411).

`/confdConfig/snmpAgent/snmpEngine/snmpEngineMaxMessageSize` (xs:nonNegativeInteger) [50000]
The maximum size of SNMP messages that the agent can send or receive. The snmpEngineMaxMessageSize is defined in the SNMP-FRAMEWORK-MIB (RFC3411).

`/confdConfig/snmpAgent/snmpEngine/authenticationFailureNotifyName` (xs:string) [""]
When the SNMP agent sends the standard authenticationFailure notification, it is delivered to the management targets defined for the snmpNotifyName in the snmpNotifyTable in SNMP-NOTIFICATION-MIB (RFC3413). If authenticationFailureNotifyName is the empty string (default), the notification is delivered to all management targets.

`/confdConfig/snmpAgent/contexts` (xs:string) [""]
A space separated list of context names which this SNMP Agent, i.e. one or more external data providers, recognize in addition to the empty context, "".

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<contexts>` items instead of a space separated list.

`/confdConfig/snmpAgent/dropWhenInUse` (xs:boolean) [false]
Whenever a set request cannot be completed, due to competing actions (typically CDB clients, or other transactions) preventing the SNMP Agent from taking the required locks on configuration stores and data providers affected by the request, the SNMP Agent will respond to the set request with an "in use" error. If dropWhenInUse is true, the SNMP Agent will silently drop the request instead.

`/confdConfig/snmpAgent/system - container element`
This section defines properties from the SNMPv2-MIB (RFC3418) for the SNMP agent.

`/confdConfig/snmpAgent/system/sysDescr` (xs:string)
A textual description of the entity. This value should include the full name and version identification of the system's hardware type, software operating-system, and networking software. The sysDescr is defined in the SNMPv2-MIB (RFC3418).

`/confdConfig/snmpAgent/system/sysObjectID` (confd:oid)
The vendor's authoritative identification of the network management subsystem contained in the entity. The sysObjectID is defined in the SNMPv2-MIB (RFC3418).

`/confdConfig/snmpAgent/system/sysServices` (xs:nonNegativeInteger) [72]
A value which indicates the set of services that this entity may potentially offer. The sysServices is defined in the SNMPv2-MIB (RFC3418).

`/confdConfig/snmpAgent/system/sysORTable - container element`
Entries that will populate the sysORTable from SNMPv2-MIB.

`/confdConfig/snmpAgent/system/sysORTable/sysOREntry - container element`
Corresponds to one entry in the sysORTable from SNMPv2-MIB.

`/confdConfig/snmpAgent/system/sysORTable/sysOREntry/sysORIndex (xs:nonNegativeInteger)`
The index for this row in the table.

`/confdConfig/snmpAgent/system/sysORTable/sysOREntry/sysORID (confd:oid)`
The OID of the AGENT-CAPABILITIES invocation.

`/confdConfig/snmpAgent/system/sysORTable/sysOREntry/sysORDescr (xs:string)`
A textual description of capabilities defined in sysORID.

`/confdConfig/snmpgw - container element`
This section defines settings which affect the behavior of the SNMP gateway.

`/confdConfig/snmpgw/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the gateway is enabled.

`/confdConfig/snmpgw/trapPort` (confd:inetPortNumber)
The port number to listen for traps on.

`/confdConfig/snmpgw/agent - container element`
Parameters for a single agent.

`/confdConfig/snmpgw/agent/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "true", the agent is enabled.

`/confdConfig/snmpgw/agent/name` (xs:token)
A name for the agent, mainly used for error reporting.

`/confdConfig/snmpgw/agent/subscriptionId` (xs:token)
The subscription id is used for indicating to which applications external traps should be sent on.

`/confdConfig/snmpgw/agent/community` (xs:string) [private]
The community string for communication with the agent. If the community string cannot be expressed in Unicode, use the element community_bin instead (see below). If both community_bin and community are specified, community is ignored.

`/confdConfig/snmpgw/agent/community_bin` (xs:hexBinary)
The community string for communication with the agent, encoded in hexBinary. For example, `<community>AB</community>` and `<community_bin>4142</community_bin>` are equivalent. The main use for this is when the community string cannot be expressed in Unicode.

`/confdConfig/snmpgw/agent/version` (v1 | v2c) [v2c]
The default protocol version to use. The value indicates the preferred version - if the agent doesn't respond, the other version will be tried.

`/confdConfig/snmpgw/agent/timeout` (xs:duration) [PT5S]
The amount of time to wait for an answer from the agent before aborting the operation. The default is five seconds.

`/confdConfig/snmpgw/agent/ip` (confd:inetAddressIP)
The host (specified as a name or an IP address) on which the agent is running.

`/confdConfig/snmpgw/agent/port` (confd:inetPortNumber) [161]
The port number to use for communication with the agent.

`/confdConfig/snmpgw/agent/module` (xs:string)
A list of MIB module names that this agent implements. Each such MIB must be convert to YANG and compiled with the --snmpgw flag to confdc.

`/confdConfig/webui - container element`
This section defines settings which decide how the embedded ConfD Web server should behave, with respect to TCP and SSL etc.

`/confdConfig/webui/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the Web server is started.

`/confdConfig/webui/serverName` (xs:string) [localhost]
The hostname the Web server serves.

`/confdConfig/webui/matchHostName`(xs:boolean) [false]
This setting specifies if the Web server only should serve URLs adhering to the serverName defined above. By default the serverName is "localhost" and matchHostName is "false", i.e. any server name can be given in the URL. If you want the server to only accept URLs adhering to the serverName, enable this setting.

`/confdConfig/webui/cacheRefreshSecs` (xs:nonNegativeInteger) [0]
The ConfD Web server uses a RAM cache for static content. An entry sits in the cache for a number of seconds before it is reread from disk (on access). The default is 0.

`/confdConfig/webui/maxRefEntries` (xs:nonNegativeInteger) [100]
Leafref and keyref entries are represented as drop-down menus in the automatically generated Web UI. By default no more than 100 entries are fetched. This element makes this number configurable.

`/confdConfig/webui/docroot`
The location of the document root on disk. If this configurable is omitted the docroot points to the next generation docroot in the ConfD distro instead.

`/confdConfig/webui/loginDir`
The loginDir element points out an alternative login directory which contains your HTML code etc used to login to the Web UI. This directory will be mapped https://<ip-address>/login. If this element is not specified the default login/ directory in the docroot will be used instead.

`/confdConfig/webui/customDir`
The custom-dir element points out a custom directory which contains the customizations you want to apply to the auto- generated Web UI. The content of this directory should be as described in the User Guide and it will be mapped to https://<ip-address>/custom.

`/confdConfig/webui/customHeaders/header`
The customHeaders element contains any number of header elements, with a valid header-field as defined in RFC7230. The headers will be part of HTTP responses on "/login.html", "/index.html" and "/jsonrpc".

`/confdConfig/webui/customHeaders/header/name` (xs:string)
RFC7230 field-name, e.g. Accept-Control-Allow-Origin

`/confdConfig/webui/customHeaders/header/value` (xs:string)
RFC7230 field-value, e.g. http://www.cisco.com

`/confdConfig/webui/disableAuth/dir`
The disableAuth element contains any number of dir elements. Each dir element points to a directory path in the docroot which should not be restricted by the AAA engine. If no dir elements are specified the following directories and files will not be restricted by the AAA engine: "/login" and "/login.html".

`/confdConfig/webui/transport - container element`
Settings deciding which transport services the Web server should listen to, e.g. TCP and SSL.

`/confdConfig/webui/transport/tcp - container element`
Settings deciding how the Web server TCP transport service should behave.

`/confdConfig/webui/transport/tcp/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the Web server uses cleart text TCP as a transport service.

`/confdConfig/webui/transport/tcp/redirect` (xs:string)
If given the user will be redirected to the specified URL. Two macros can be specified, i.e. @HOST@ and @PORT@. For example https://@HOST@:443 or https://192.12.4.3:@PORT@

`/confdConfig/webui/transport/tcp/disableNonAuthRedirect` (xs:boolean) [false]
disableNonAuthRedirect is either "true" or "false". If "true" non-authenticated HTTP requests (expect "/" and "/index.html") result in a 404 HTTP reply. If false all non-authenticated requests are redirected to "/login.html".

`/confdConfig/webui/transport/tcp/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the Web server should listen to. 0.0.0.0 means that it listens to the port (/confdConfig/webui/transport/tcp/port) for all IPv4 addresses on the machine.

`/confdConfig/webui/transport/tcp/port` (confd:inetPortNumber) [8008]
port is a valid port number to be used in combination with /confdConfig/webui/transport/tcp/ip.

`/confdConfig/webui/transport/tcp/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the Web server also listens to for TCP connections. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/webui/transport/tcp/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/webui/transport/tcp/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the Web server for TCP connections.

`/confdConfig/webui/transport/ssl - container element`
Settings deciding how the Web server SSL (Secure Sockets Layer) transport service should behave.

SSL is widely deployed on the Internet and virtually all bank transactions as well as all on-line shopping today is done with SSL encryption. There are many good sources on describing SSL in detail, e.g. http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/ which describes how to manage certificates and keys.

`/confdConfig/webui/transport/ssl/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the Web server uses SSL as a transport service.

`/confdConfig/webui/transport/ssl/disableNonAuthRedirect` (xs:boolean) [false]
disableNonAuthRedirect is either "true" or "false". If "true" non-authenticated HTTP requests (expect "/" and "/index.html") result in a 404 HTTP reply. If false all non-authenticated requests are redirected to "/login.html".

`/confdConfig/webui/transport/ssl/redirect` (xs:string)
If given the user will be redirected to the specified URL. Two macros can be specified, i.e. @HOST@ and @PORT@. For example http://@HOST@:80 or http://192.12.4.3:@PORT@

`/confdConfig/webui/transport/ssl/ip` (confd:inetAddressIP) [0.0.0.0]
ip is an IP address which the Web server should listen to. 0.0.0.0 means that it listens to the port (/confdConfig/webui/transport/ssl/port) for all IPv4 addresses on the machine.

`/confdConfig/webui/transport/ssl/port` (confd:inetPortNumber) [8888]
port is a valid port number to be used in combination with /confdConfig/webui/transport/ssl/ip.

`/confdConfig/webui/transport/ssl/extraIpPorts` (ip:port ip:port ...) []
extraIpPorts is a space separated list of ip:port pairs which the Web server also listens to for SSL connections. For IPv6 addresses, the syntax [ip]:port may be used. If the ":port" is omitted, `/confdConfig/webui/transport/ssl/port` is used. Example: "10.45.22.11:4777 127.0.0.1 :::88 [::]"

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<extraIpPorts>` items instead of a space separated list. Example:

```
<extraIpPorts>10.45.22.11:4777</extraIpPorts>
<extraIpPorts>127.0.0.1</extraIpPorts>
<extraIpPorts>:::88</extraIpPorts>
<extraIpPorts>[::]</extraIpPorts>
```

`/confdConfig/webui/transport/ssl/dscp` (xs:unsignedByte)
Support for setting the Differentiated Services Code Point (6 bits) for traffic originating from the Web server for SSL connections.

`/confdConfig/webui/transport/ssl/keyFile` (xs:string)
Specifies which file that contains the private key for the certificate. Read more about certificates in /confdConfig/webui/transport/ssl/certFile. If this configurable is omitted the keyFile points to a built-in self signed certificate/key in the ConfD distro instead. Note: Only use this certificate/key for test purposes.

`/confdConfig/webui/transport/ssl/certFile` (xs:string)
Specifies which file that contains the server certificate. The certificate is either a self-signed test certificate or a genuin and validated certificate bought from a CA (Certificate Authority). If this configurable is omitted the keyFile points to a built-in self signed certificate/key in the ConfD distro instead. Note: Only use this certificate/key for test purposes.

The ConfD distribution comes with a server certificate which can be used for testing purposes ($CONFD_DIST/var/confd/webui/cert/host.{cert,key}). This server certificate has been generated using a local CA certificate:

```
$ openssl
OpenSSL> genrsa -out ca.key 4096
OpenSSL> req -new -x509 -days 3650 -key ca.key -out ca.cert
OpenSSL> genrsa -out host.key 4096
```

```
OpenSSL> req -new -key host.key -out host.csr
OpenSSL> x509 -req -days 365 -in host.csr -CA ca.cert \
             -CAkey ca.key -set_serial 01 -out host.cert
```

`/confdConfig/webui/transport/ssl/caCertFile` (xs:string)
Specifies which file that contains the trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable CA certificates passed to the client when a certificate is requested.

The ConfD distribution comes with a CA certificate which can be used for testing purposes ($CONFD_DIST/var/confd/webui/ca_cert/ca.cert).

`/confdConfig/webui/transport/ssl/verify` (1 | 2 | 3) [1]
Specifies the level of verification the server does on client certificates. 1 means nothing, 2 means the server will ask the client for a certificate but not fail if the client does not supply a client certificate, 3 means that the server requires the client to supply a client certificate.

If caCertFile has been set to the ca.cert file generated above you can verify that it works correctly using, for example:

```
$ openssl s_client -connect 127.0.0.1:8888 \
                   -cert client.cert -key client.key
```

For this to work client.cert must have been generated using the ca.cert from above:

```
OpenSSL> genrsa -out client.key 4096
OpenSSL> req -new -key client.key -out client.csr
OpenSSL> x509 -req -days 3650 -in client.csr -CA ca.cert \
             -CAkey ca.key -set_serial 01 -out client.cert
```

`/confdConfig/webui/transport/ssl/depth` (xs:nonNegativeInteger) [1]
Specifies the depth of certificate chains the server is prepared to follow when verifying client certificates.

`/confdConfig/webui/transport/ssl/ciphers` (xs:string) [DEFAULT]
Specifies the cipher suites to be used by the server as a colon-separated list from the set DHE-RSA-AES256-SHA256, DHE-DSS-AES256-SHA256, AES256-SHA256, DHE-RSA-AES128-SHA256, DHE-DSS-AES128-SHA256, AES128-SHA256, DHE-RSA-AES256-SHA, DHE-DSS-AES256-SHA, AES256-SHA, EDH-RSA-DES-CBC3-SHA, EDH-DSS-DES-CBC3-SHA, DES-CBC3-SHA, DHE-RSA-AES128-SHA, DHE-DSS-AES128-SHA, AES128-SHA, RC4-SHA, RC4-MD5, EDH-RSA-DES-CBC-SHA, and DES-CBC-SHA, or the word "DEFAULT" (use all supported cipher suites in that list). See the OpenSSL manual page ciphers(1) for the definition of the cipher suites. NOTE: The general cipher list syntax described in ciphers(1) is not supported.

`/confdConfig/webui/transport/ssl/protocols` (xs:string) [DEFAULT]
Specifies the SSL/TLS protocol versions to be used by the server as a whitespace-separated list from the set sslv3 tlsv1 tlsv1.1 tlsv1.2, or the word "DEFAULT" (use all supported protocol versions except sslv3).

`/confdConfig/webui/cgi - container element`
CGI-script support

`/confdConfig/webui/cgi/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", CGI-script support is enabled.

`/confdConfig/webui/cgi/dir` (xs:string)
The directory path to the location of the CGI-scripts.

`/confdConfig/webui/idleTimeout` (xs:duration) [PT30M]
Maximum idle time before terminating a Web UI session. Default is PT30M, ie 30 minutes. PT0M means no timeout.

`/confdConfig/webui/namedCommands - container element`
Named commands are used to define a well defined set of executables which can be run on the target device from the browser using a call to Maapi.runCmd().

`/confdConfig/webui/namedCommands/exec`
"exec" directives specify how a named command is mapped to an executable or a shell script. It contains (in order) one "name" directive, one "osCommand" directive, zero or one "args" directives and zero or one "options" directives.

An example:

```
<exec name="cli">
  <osCommand>confd_cli</osCommand>
  <args>-u$(user) --proto http</args>
</exec>
```

`/confdConfig/webui/namedCommands/exec/name` (xs:token)
The command alias name to be used in Maapi.runCmd() calls.

`/confdConfig/webui/namedCommands/exec/osCommand` (xs:token)
The "osCommand" directive specifies the path to the executable or shell script to be called. If the command is in the $PATH (as specified when we start the ConfD daemon) the path may just be the name of the command.

The command is invoked as if it had been executed by exec(3), i.e. not in a shell environment such as "/bin/sh -c ...".

`/confdConfig/webui/namedCommands/exec/args` (argsType)
The "args" directive specifies the arguments to use when executing the command specified by the "osCommand" directive. argsType is a space-separated list of argument strings.

`/confdConfig/webui/namedCommands/exec/options - container element`
The "options" directive specifies how the command is be executed. It contains (in any order) zero or one "uid" directives and zero or one "wd" directives.

`/confdConfig/webui/namedCommands/exec/options/uid` (idType)
The "uid" directive specifies which user id to use when executing the command. Possible values are:

`confd` (default)	The command is run as the same user id as the ConfD daemon.
`user`	The command is run as the same user id as the user logged in to the CLI, i.e. we have to make sure that this user id exists as an actual user id on the device.
`root`	The command is run as root.
`<uid>` (the numerical user *<uid>*)	The command is run as the user id <uid>.

Note

If uid is set to either "user", "root" or "<uid>" the the ConfD daemon must have been started as root (or

setuid), or the cmdptywrapper must have setuid root permissions.

`/confdConfig/rest - container element`
This section defines settings for the RESTful API to ConfD.

`/confdConfig/rest/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the RESTful API is activated. Currently, since the REST API uses the same Web server as the Web UI, the Web UI muse also be enabled, see `/confdConfig/webui/enabled`.

`/confdConfig/rest/showHidden` (xs:boolean) [false]
Hidden nodes are not shown by default in REST. Such nodes can be unhidden to the REST client by including the query parameter "unhide", which is a comma separated list of

```
<hide-group-name>[;<passwd>]
```

If showHidden is set to "true", hidden nodes are always shown in the REST API.

`/confdConfig/rest/customHeaders/header - container element`
The customHeaders element contains any number of header elements, with a valid header-field as defined in RFC7230 3.2.

`/confdConfig/rest/customHeaders/header/name` (xs:string)
RFC7230 field-name, e.g. Accept-Control-Allow-Origin

`/confdConfig/rest/customHeaders/header/value` (xs:string)
RFC7230 field-value, e.g. http://www.cisco.com

`/confdConfig/subagents - container element`
Present only if ConfD runs as a master agent. Lists all registered subagents.

`/confdConfig/subagents/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", ConfD acts as a master agent.

`/confdConfig/subagents/subagent - container element`
Parameters for a single subagent.

`/confdConfig/subagents/subagent/name` (xs:token)
The name of the subagent. It is used as a unique identifier of the subagent. The name is included in the subagent events (see confd_lib_events(3)) generated by ConfD.

`/confdConfig/subagents/subagent/enabled` (xs:boolean) [true]
enabled is either "true" or "false". If "false", the subagent is ignored by ConfD.

`/confdConfig/subagents/subagent/tcp - container element`
Parameters to be used when the masteragent communicates with the subagent over plain text TCP. This is more efficient than SSH, but TCP is non-standard.

`/confdConfig/subagents/subagent/tcp/ip` (confd:inetAddressIP)
The IP address where the subagent listens for NETCONF TCP connections.

`/confdConfig/subagents/subagent/tcp/port` (confd:inetPortNumber) [2023]
The port where the subagent listens for NETCONF TCP connections.

`/confdConfig/subagents/subagent/tcp/confdAuth - container element`
Currently, in order to use NETCONF over TCP, the subagent must understand the ConfD proprietary TCP header, described in the NETCONF chapter in the ConfD User's Guide.

`/confdConfig/subagents/subagent/tcp/confdAuth/user` (xs:string)
The user name to be used for authorization on the subagent.

`/confdConfig/subagents/subagent/tcp/confdAuth/group` (xs:string)
The group name to be used for authorization on the subagent.

`/confdConfig/subagents/subagent/ssh - container element`
Parameters to be used when the masteragent communicates with the subagent over SSH.

`/confdConfig/subagents/subagent/ssh/ip` (confd:inetAddressIP)
The IP address where the subagent listens for NETCONF SSH connections.

`/confdConfig/subagents/subagent/ssh/port` (confd:inetPortNumber) [2022]
The port where the subagent listens for NETCONF SSH connections.

`/confdConfig/subagents/subagent/ssh/user` (xs:string)
The SSH user name used for authentication at the subagent.

`/confdConfig/subagents/subagent/ssh/password` (xs:string)
The SSH user's password.

`/confdConfig/subagents/subagent/mount - container element`
This parameter defines where in the data hierarchy the subagent is registered. It consists of a path which must exist in the data model of the master agent, and the name of the node which the subagent implements.

`/confdConfig/subagents/subagent/mount/path` (xs:string)
The path, in restricted XPath syntax, where the subagent's data is mounted. The XPath is restricted as an instance-identifier (See confd_types(3)). To mount on the top level, use "/". Note that the XPath expression must not contain any namespace prefixes.

If the subagent mounts more than one node, this object is a space separated list of paths.

`/confdConfig/subagents/subagent/mount/node` (xs:QName)
The namespace and name of the top-level node in that namespace.

If the subagent mounts more than one node, this object is a space separated list of QNames, which must be of the same length as the 'path' object.

Note

When the `confd_dyncfg.yang` YANG module is used, this element is a `leaf-list`, i.e. multiple values are represented by multiple `<node>` items instead of a space separated list.

`/confdConfig/subagents/subagent/disableSubtreeOptimization` (xs:boolean) [false]
Whenever possible, the master agent sends a single subtree filter request, instead of one request for each object. If the subagent cannot handle these requests, for any reason, set this parameter to "true".

`/confdConfig/notifications - container element`
This section defines settings which affect notifications.

`/confdConfig/notifications/eventStreams - container element`
Lists all available notification event streams.

`/confdConfig/notifications/eventStreams/stream - container element`
Parameters for a single notification event stream.

`/confdConfig/notifications/eventStreams/stream/name` (xs:string)
The name attached to a specific event stream.

`/confdConfig/notifications/eventStreams/stream/description` (xs:string)
A descriptive text attached to a specific event stream.

`/confdConfig/notifications/eventStreams/stream/replaySupport` (xs:boolean)
Signals if replay support is available for a specific event stream.

`/confdConfig/notifications/eventStreams/stream/builtinReplayStore - container element`
Parameters for the builtin replay support for this event stream.

If replay support is enabled ConfD automatically stores all notifications on disk ready to be replayed should a NETCONF manager ask for logged notifications. The replay store uses a set of wrapping log files on disk (of a certain number and size) to store the notifications.

The max size of each wrap log file (see below) should not be too large. This to achieve fast replay of notifications in a certain time range. If possible use a larger number of wrap log files instead.

If in doubt use the recommended settings (see below).

`/confdConfig/notifications/eventStreams/ stream/builtinReplayStore/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "false", the applications must implement its own replay support.

`/confdConfig/notifications/eventStreams/ stream/builtinReplayStore/dir` (xs:string)
The wrapping log files will be put in this disk location.

`/confdConfig/notifications/eventStreams/ stream/builtinReplayStore/maxSize` (confd:size)
The max size of each log wrap file. The recommended setting is ~S10MB.

`/confdConfig/notifications/eventStreams/ stream/builtinReplayStore/maxFiles` (xs:integer)
The max number of log wrap files. The recommended setting is ~50 files.

`/confdConfig/opcache - container element`
This section defines settings which affect the behavior of the operational data cache - see the "Operational data" chapter in the User Guide.

`/confdConfig/opcache/enabled` (xs:boolean) [false]
enabled is either "true" or "false". If "true", the cache is enabled.

`/confdConfig/opcache/timeout` (xs:nonNegativeInteger)
The amount of time to keep data in the cache, in seconds.

SEE ALSO

confd(1) - command to start and control the ConfD daemon

confd_lib_dp(3) - callback library for connecting to ConfD

confd_types(3) - Confd XML value representation in C

confd_cfg.xsd - A W3C XML schema (`http://tail-f.com/ns/confd_cfg/1.0`) describing the daemon configuration.

`confd.conf.example` - A commented confd.conf example file.

Name

mib_annotations — MIB annotations file format

DESCRIPTION

This manual page describes the syntax and semantics used to write MIB annotations. A MIB annotation file is used to modify the behavior of certain MIB objects without having to edit the original MIB file.

MIB annotations are separate file with a .miba suffix, and is applied to to a MIB when a YANG module is generated and when the MIB is compiled. See confdc(1).

SYNTAX

Each line in a MIB annotation file has the following syntax:

```
<MIB Object Name> <modifier> [= <value>]
```

where `modifier` is one of `max_access`, `display_hint`, `behavior`, `unique`, or `operational`.

Blank lines are ignored, and lines starting with # are treated as comments and ignored.

If `modifier` is `max_access`, `value` must be one of `not_accessible` or `read_only`.

If `modifier` is `display_hint`, `value` must be a valid DISPLAY-HINT value. The display hint is used to determine if a string object should be treated as text or binary data.

If `modifier` is `behavior`, `value` must be one of `noSuchObject` or `noSuchInstance`. When a YANG module is generated from a MIB, objects with a specified behavior are not converted to YANG. When the SNMP agent responds to SNMP requests for such an object, the corresponding error code is used.

If `modifier` is `unique`, `value` must be a valid YANG "unique" expression, i.e., a space-separated list of column names. This modifier must be given on table entries.

If `modifier` is `operational`, there must not be any `value` given. A writable object marked as `operational` will be translated into a non-configuration YANG node, marked with a `tailf:writable true` statement, indicating that the object represents writable operational data.

EXAMPLE

An example of a MIB annotation file.

```
# the following object does not have value
ifStackLastChange behavior = noSuchInstance

# this deprecated table is not implemented
ifTestTable behavior = noSuchObject
```

SEE ALSO

The ConfD User Guide

confdc(1) YANG compiler

Name

tailf_yang_cli extensions — Tail-f YANG CLI extensions

Synopsis

```
tailf:cli-add-mode

tailf:cli-allow-join-with-key

tailf:cli-allow-join-with-value

tailf:cli-allow-key-abbreviation

tailf:cli-allow-range

tailf:cli-allow-wildcard

tailf:cli-autowizard

tailf:cli-boolean-no

tailf:cli-break-sequence-commands

tailf:cli-case-insensitive

tailf:cli-case-sensitive

tailf:cli-column-align

tailf:cli-column-stats

tailf:cli-column-width

tailf:cli-compact-stats

tailf:cli-compact-syntax

tailf:cli-completion-actionpoint

tailf:cli-configure-mode

tailf:cli-custom-error

tailf:cli-custom-range

tailf:cli-custom-range-actionpoint

tailf:cli-custom-range-enumerator

tailf:cli-delayed-auto-commit

tailf:cli-delete-container-on-delete

tailf:cli-delete-when-empty

tailf:cli-diff-dependency

tailf:cli-disabled-info
```

```
tailf:cli-disallow-value
tailf:cli-display-empty-config
tailf:cli-display-separated
tailf:cli-drop-node-name
tailf:cli-embed-no-on-delete
tailf:cli-enforce-table
tailf:cli-exit-command
tailf:cli-explicit-exit
tailf:cli-expose-key-name
tailf:cli-expose-ns-prefix
tailf:cli-flat-list-syntax
tailf:cli-flatten-container
tailf:cli-full-command
tailf:cli-full-no
tailf:cli-full-show-path
tailf:cli-hide-in-submode
tailf:cli-ignore-modified
tailf:cli-incomplete-command
tailf:cli-incomplete-no
tailf:cli-incomplete-show-path
tailf:cli-instance-info-leafs
tailf:cli-key-format
tailf:cli-list-syntax
tailf:cli-min-column-width
tailf:cli-mode-name
tailf:cli-mode-name-actionpoint
tailf:cli-mount-point
tailf:cli-multi-line-prompt
tailf:cli-multi-value
tailf:cli-multi-word-key
```

```
tailf:cli-no-key-completion
tailf:cli-no-keyword
tailf:cli-no-match-completion
tailf:cli-no-name-on-delete
tailf:cli-no-value-on-delete
tailf:cli-oper-info
tailf:cli-operational-mode
tailf:cli-optional-in-sequence
tailf:cli-prefix-key
tailf:cli-preformatted
tailf:cli-range-delimiters
tailf:cli-range-list-syntax
tailf:cli-recursive-delete
tailf:cli-remove-before-change
tailf:cli-replace-all
tailf:cli-reset-container
tailf:cli-run-template
tailf:cli-run-template-enter
tailf:cli-run-template-footer
tailf:cli-run-template-legend
tailf:cli-sequence-commands
tailf:cli-show-config
tailf:cli-show-long-obu-diffs
tailf:cli-show-no
tailf:cli-show-obu-comments
tailf:cli-show-order-tag
tailf:cli-show-order-taglist
tailf:cli-show-template
tailf:cli-show-template-enter
tailf:cli-show-template-footer
```

```
tailf:cli-show-template-legend

tailf:cli-show-with-default

tailf:cli-strict-leafref

tailf:cli-suppress-key-abbreviation

tailf:cli-suppress-key-sort

tailf:cli-suppress-list-no

tailf:cli-suppress-mode

tailf:cli-suppress-no

tailf:cli-suppress-range

tailf:cli-suppress-shortenabled

tailf:cli-suppress-show-conf-path

tailf:cli-suppress-show-match

tailf:cli-suppress-show-path

tailf:cli-suppress-silent-no

tailf:cli-suppress-table

tailf:cli-suppress-validation-warning-prompt

tailf:cli-suppress-wildcard

tailf:cli-table-footer

tailf:cli-table-legend

tailf:cli-trim-default

tailf:cli-value-display-template
```

DESCRIPTION

This manpage describes all the Tail-f CLI extension statements.

The YANG source file `$CONFD_DIR/src/confd/yang/tailf-cli-extensions.yang` gives the exact YANG syntax for all Tail-f YANG CLI extension statements - using the YANG language itself.

Most of the concepts implemented by the extensions listed below are described in the User Guide.

YANG STATEMENTS

tailf:cli-add-mode

Creates a mode of the container.

Can be used in config nodes only.

Used in I- and C-style CLIs.

The *cli-add-mode* statement can be used in: *container*, *tailf:symlink*, and *refine*.

tailf:cli-allow-join-with-key

Indicates that the list name may be written together with the first key, without requiring a whitespace in between, ie allowing both interface ethernet1/1 and interface ethernet 1/1

Used in I- and C-style CLIs.

The *cli-allow-join-with-key* statement can be used in: *list*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-display-joined Specifies that the joined version should be used when displaying the configuration in C- and I- mode.

tailf:cli-allow-join-with-value

Indicates that the leaf name may be written together with the value, without requiring a whitespace in between, ie allowing both interface ethernet1/1 and interface ethernet 1/1

Used in I- and C-style CLIs.

The *cli-allow-join-with-value* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-display-joined Specifies that the joined version should be used when displaying the configuration in C- and I- mode.

tailf:cli-allow-key-abbreviation

Key values can be abbreviated.

In the J-style CLI this is relevant when using the commands 'delete' and 'edit'.

In the I- and C-style CLIs this is relevant when using the commands 'no', 'show configuration' and for commands to enter submodes.

See also /confdConfig/cli/allowAbbrevKeys in confd.conf(5).

The *cli-allow-key-abbreviation* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-allow-range

Means that the non-integer key should allow range expressions.

Can be used in key leafs only.

Used in J-, I- and C-style CLIs.

The *cli-allow-range* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-allow-wildcard

Means that the list allows wildcard expressions in the 'show' pattern.

See also /confdConfig/cli/allowWildcard in confd.conf(5).

Used in J-, I- and C-style CLIs.

The *cli-allow-wildcard* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-autowizard

Specifies that the autowizard should include this leaf even if the leaf is optional.

One use case is when implementing pre-configuration of devices. A config false node can be defined for showing if the configuration is active or not (preconfigured).

Used in J-, I- and C-style CLIs.

The *cli-autowizard* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-boolean-no

Specifies that a leaf of type boolean should be displayed as '<leafname>' if set to true, and 'no <leafname>' if set to false.

Cannot be used in conjunction with tailf:cli-hide-in-submode or tailf:cli-compact-syntax.

Used in I- and C-style CLIs.

The *cli-boolean-no* statement can be used in: *typedef*, *leaf*, *refine*, and *tailf:symlink*.

The following substatements can be used:

tailf:cli-reversed Specified that true should be displayed as 'no <name>' and false as 'name'.

Used in I- and C-style CLIs.

tailf:cli-break-sequence-commands

Specifies that previous cli-sequence-command declaration should stop at this point. Only applicable when a cli-sequence-command declaration has been used in the parent container.

Used in I- and C-style CLIs.

The *cli-break-sequence-commands* statement can be used in: *leaf*, *tailf:symlink*, *leaf-list*, *list*, *container*, and *refine*.

tailf:cli-case-insensitive

Specifies that node is case-insensitive. If applied to a container or a list, any nodes below will also be case-insensitive.

Note that this will override any case-insensitivity settings configured in confd.conf

The *cli-case-insensitive* statement can be used in: *container*, *list*, and *leaf*.

tailf:cli-case-sensitive

Specifies that this node is case-sensitive. If applied to a container or a list, any nodes below will also be case-sensitive.

Note that this will override any case-sensitivity settings configured in confd.conf

The *cli-case-sensitive* statement can be used in: *container*, *list*, and *leaf*.

tailf:cli-column-align *value*

Specifies the alignment of the data in the column in the auto-rendered tables.

Used in J-, I- and C-style CLIs.

The *cli-column-align* statement can be used in: *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-column-stats

Display leafs in the container as columns, i.e., do not repeat the name of the container on each line, but instead indent each leaf under the container.

Used in I- and C-style CLIs.

The *cli-column-stats* statement can be used in: *container*, *refine*, and *tailf:symlink*.

tailf:cli-column-width *value*

Set a fixed width for the column in the auto-rendered tables.

Used in J-, I- and C-style CLIs.

The *cli-column-width* statement can be used in: *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-compact-stats

Instructs the CLI engine to use the compact representation for this node. The compact representation means that all leaf elements are shown on a single line.

Used in J-, I- and C-style CLIs.

The *cli-compact-stats* statement can be used in: *list*, *container*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-wrap If present, the line will be wrapped at screen width.

tailf:cli-width Specifies a fixed terminal width to use before wrapping line. It is only used when tailf:cli-wrap is present. If a width is not specified the line is wrapped when the terminal width is reached.

tailf:cli-delimiter Specifies a string to print between the leaf name and its value when displaying leaf values.

tailf:cli-prettify If present, dashes (-) and underscores (_) in leaf names are replaced with spaces.

tailf:cli-spacer Specifies a string to print between the nodes.

tailf:cli-compact-syntax

Instructs the CLI engine to use the compact representation for this node in the 'show running-configuration' command. The compact representation means that all leaf elements are shown on a single line.

Cannot be used in conjunction with tailf:cli-boolean-no.

Used in I- and C-style CLIs.

The *cli-compact-syntax* statement can be used in: *list*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-completion-actionpoint *value*

Specifies that completion for the leaf values is done through a callback function.

The argument is the name of an actionpoint, which must be implemented by custom code. In the action-point, the completion() callback function will be invoked. See confd_lib_dp(3) for details.

Used in J-, I- and C-style CLIs.

The *cli-completion-actionpoint* statement can be used in: *leaf-list*, *leaf*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-completion-id Specifies a string which is passed to the callback when invoked. This makes it possible to use the same callback at several locations and still keep track of which point it is invoked from.

tailf:cli-configure-mode

An action or rpc with this attribute will be available in configure mode, but not in operational mode.

The default is that the action or rpc is available in both configure and operational mode.

Used in J-, I- and C-style CLIs.

The *cli-configure-mode* statement can be used in: *tailf:action* and *rpc*.

tailf:cli-custom-error *text*

This statement specifies a custom error message to be displayed when the user enters an invalid value.

The *cli-custom-error* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-custom-range

Specifies that the key should support ranges. A type matching the range expression must be supplied.

Can be used in key leafs only.

Used in J-, I- and C-style CLIs.

The *cli-custom-range* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-range-type This statement contains the name of a derived type, possibly with a prefix. If no prefix is given, the type must be defined in the local module. For example:

cli-range-type p:my-range-type;

All range expressions must match this type, and a valid key value must not match this type.

tailf:cli-custom-range-actionpoint *value*

Specifies that the list supports range expressions and that a custom function will be invoked to determine if an instance belong in the range or not. At least one key element needs a cli-custom-range statement.

The argument is the name of an actionpoint, which must be implemented by custom code. In the action-point, the completion() callback function will be invoked. See confd_lib_dp(3) for details.

When a range expression value which matches the type is given in the CLI, the CLI engine will invoke the callback with each existing list entry instance. If the callback returns CONFD_OK, it matches the range expression, and if it returns CONFD_ERR, it doesn't match.

Used in J-, I- and C-style CLIs.

The *cli-custom-range-actionpoint* statement can be used in: *list*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-completion-id Specifies a string which is passed to the callback when invoked. This makes it possible to use the same callback at several locations and still keep track of which point it is invoked from.

tailf:cli-allow-caching Allow caching of the evaluation results between different parent paths.

tailf:cli-custom-range-enumerator *value*

Specifies a callback to invoke to get an array of instances matching a regular expression. This is used when instances should be allowed to be created using a range expression in set.

The callback is not used for delete or show operations.

The callback is allowed to return a superset of all matching instances since the instances will be filtered using the range expression afterwards.

Used in J-, I- and C-style CLIs.

The *cli-custom-range-enumerator* statement can be used in: *list*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-completion-id Specifies a string which is passed to the callback when invoked. This makes it possible to use the same callback at several locations and still keep track of which point it is invoked from.

tailf:cli-allow-caching Allow caching of the evaluation results between different parent paths.

tailf:cli-delayed-auto-commit

Enables transactions while in a specific submode (or submode of that mode). The modifications performed in that mode will not take effect until the user exits that submode.

Can be used in config nodes only. If used in a container, the container must also have a tailf:cli-add-mode statement, and if used in a list, the list must not also have a tailf:cli-suppress-mode statement.

Used in I- and C-style CLIs.

The *cli-delayed-auto-commit* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-delete-container-on-delete

Specifies that the parent container should be deleted when . this leaf is deleted.

The *cli-delete-container-on-delete* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-delete-when-empty

Instructs the CLI engine to delete the list when the last list instance is deleted'. Requires that cli-suppress-mode is set.

The behavior is recursive. If all optional leafs in a list instance are deleted the list instance itself is deleted. If that list instance happens to be the last list instance in a list it is also deleted. And so on. Used in I- and C-style CLIs.

The *cli-delete-when-empty* statement can be used in: *list* and *container*.

tailf:cli-diff-dependency *path*

Tells the 'show configuration' command, and the diff generator that this node depends on another node. When removing the node with this declaration, it should be removed before the node it depends on is removed, ie the declaration controlls the ordering of the commands in the 'show configuration' output.

Applies to C-style

The *cli-diff-dependency* statement can be used in: *container*, *list*, *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:xpath-root

tailf:cli-trigger-on-set Specify that the dependency should trigger on set/modify of the target path, but deletion of the target will trigger the current node to be placed in front of the target.

The annotation can be used to get the diff behavior where one leaf is first deleted before the other leaf is set. For example, having the data model below:

container X { leaf A { tailf:cli-diff-dependency "../B" { tailf:cli-trigger-on-set; } type empty; } leaf B { tailf:cli-diff-dependency "../A" { tailf:cli-trigger-on-set; } type empty; } }

produces the following diffs when setting one leaf and deleting the other

no X A X B

and

no X B X A

this can also be done with list instances, for example

list a { key id;

leaf id { tailf:cli-diff-dependency "/c[id=current()/../id]" { tailf:cli-trigger-on-set; } type string; } }

list c { key id; leaf id { tailf:cli-diff-dependency "/a[id=current()/../id]" { tailf:cli-trigger-on-set; } type string; } }

we get

no a foo c foo !

and

no c foo a foo !

In the above case if we have the same id in list "a" and "c" and we delete the instance in one list, and add it in the other, then the deletion will always preceed the create.

tailf:cli-trigger-on-delete This annotation can be used togeter with tailf:cli-trigger-on-set to also get the behavior that when deleting the target display changes to this node first. For exmaple:

container settings { tailf:cli-add-mode;

leaf opmode { tailf:cli-no-value-on-delete;

type enumeration { enum nat; enum transparent; } }

leaf manageip { when "../opmode = 'transparent'"; mandatory true; tailf:cli-no-value-on-delete; tailf:cli-diff-dependency '../opmode' { tailf:cli-trigger-on-set; tailf:cli-trigger-on-delete; }

type string; } }

What we are trying to achieve here is that if manageip is deleted, it should be displayed before opmode, but if we configure both opmode and manageip, we should display opmode first, ie get the diffs:

settings opmode transparent manageip 1.1.1.1 !

and

settings no manageip opmode nat !

and

settings no manageip no opmode !

The cli-trigger-on-set annotation will cause the 'no manageip' command to be displayed before setting opmode. The tailf:cli-trigger-on-delete will cause 'no manageip' to be placed before 'no opmode' when both are deleted.

In the first diff where both are created, opmode will come first due to the diff-dependency setting, regardless of the cli-trigger-on-delete and cli-trigger-on-set.

tailf:cli-trigger-on-all Specify that the dependency should always trigger. It is the same as placing one element before another in the data model. For example, given the data model:

container X { leaf A { tailf:cli-diff-dependency '../B' { tailf:cli-trigger-on-all; } type empty; } leaf B { type empty; } }

We get the diffs

X B X A

and

no X B no X A

tailf:cli-disabled-info *value*

Specifies an info string that will be used as a descriptive text for the value 'disable' (false) of boolean-typed leafs when the confd.conf(5) setting /confdConfig/cli/useShortEnabled is set to 'true'.

Used in J-, I- and C-style CLIs.

The *cli-disabled-info* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-disallow-value *value*

Specifies that a pattern for invalid values.

Used in I- and C-style CLIs.

The *cli-disallow-value* statement can be used in: *leaf*, *leaf-list*, *refine*, and *tailf:symlink*.

tailf:cli-display-empty-config

Specifies that the node will be included when doing a 'show stats', even if it is a non-config node, provided that the list contains at least one non-config node.

Used in J-style CLI.

The *cli-display-empty-config* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-display-separated

Tells CLI engine to display this container as a separate line item even when it has children. Only applies to presence containers.

Applicable for optional containers in the C- and I- style CLIs.

The *cli-display-separated* statement can be used in: *container*, *tailf:symlink*, and *refine*.

tailf:cli-drop-node-name

Specifies that the name of a node is not present in the CLI.

If tailf:cli-drop-node-name is given on a child to a list node, we recommend that you also use tailf:cli-suppress-mode on that list node, otherwise the CLI will be very confusing.

For example, consider this data model, from the tailf-aaa module:

```
list alias {
 key name;
 leaf name {
   type string;
 }
 leaf expansion {
   type string;
   mandatory true;
   tailf:cli-drop-node-name;
 }
}
```

If you type 'alias foo' in the CLI, you would end up in the 'alias' submode. But since the expansion is dropped, you would end up specifying the expansion value without typing any command.

If, on the other hand, the 'alias' list had a tailf:cli-suppress-mode statement, you would set an expansion 'bar' by typing 'alias foo bar'.

tailf:cli-drop-node-name cannot be used inside tailf:action.

Used in I- and C-style CLIs.

The *cli-drop-node-name* statement can be used in: *leaf*, *container*, *list*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-embed-no-on-delete

Embed no in front of the element name insead of at the beginning of the line.

Applies to C-style

The *cli-embed-no-on-delete* statement can be used in: *leaf*, *container*, *list*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-enforce-table

Forces the generation of a table for a list element node regardless of whether the table will be too wide or not. This applies to the tables generated by the auto-rendred show commands for non-config data.

Used in I- and C-style CLIs.

The *cli-enforce-table* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-exit-command *value*

Tells the CLI to add an explicit exit-from-submode command. The tailf:info substatement can be used for adding a custom info text for the command.

Used in I- and C-style CLIs.

The *cli-exit-command* statement can be used in: *list*, *container*, *refine*, and *tailf:symlink*.

The following substatements can be used:

tailf:info

tailf:cli-explicit-exit

Tells the CLI to add an explicit exit command when displaying the configuration. It will not be added if cli-exit-command is defined as well. The annotation is inherited by all sub-modes.

Used in I- and C-style CLIs.

The *cli-explicit-exit* statement can be used in: *list*, *container*, *refine*, and *tailf:symlink*.

tailf:cli-expose-key-name

Force the user to enter the name of the key and display the key name when displaying the running-configuration.

Used in I- and C-style CLIs.

The *cli-expose-key-name* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-expose-ns-prefix

When used force the CLI to display namespace prefix of all children.

The *cli-expose-ns-prefix* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-flat-list-syntax

Specifies that elements in a leaf-list should be entered without surrounding brackets. Also, multiple elements can be added to a list or deleted from a list.

Used in J-, I- and C-style CLIs.

The *cli-flat-list-syntax* statement can be used in: *leaf-list* and *refine*.

The following substatements can be used:

tailf:cli-replace-all

tailf:cli-flatten-container

Allows the CLI to exit the container and continue to input from the parent container when all leaves in the current container has been set.

Can be used in config nodes only.

Used in I- and C-style CLIs.

The *cli-flatten-container* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-full-command

Specifies that an auto-rendered command should be considered complete, ie, no additional leaves or containers can be entered on the same command line.

Used in I- and C-style CLIs.

The *cli-full-command* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-full-no

Specifies that an auto-rendered 'no'-command should be considered complete, ie, no additional leaves or containers can be entered on the same command line.

Used in I- and C-style CLIs.

The *cli-full-no* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-full-show-path

Specifies that a path to the show command is considered complete, i.e., no more elements can be added to the path. It can also be used to specify a maximum number of keys to be given for lists.

Used in J-, I- and C-style CLIs.

The *cli-full-show-path* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-max-keys Specifies the maximum number of allowed keys for the show command.

tailf:cli-hide-in-submode

Hide leaf when submode has been entered. Mostly useful when leaf has to be entered in order to enter a submode. Also works for flattened containers.

Cannot be used in conjunction with tailf:cli-boolean-no.

Used in I- and C-style CLIs.

The *cli-hide-in-submode* statement can be used in: *leaf*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-ignore-modified

Tells the cdb_cli_diff_iterate system call to not generate a CLI string when this container is modified. The string will instead be generated for the modified sub-element.

Applies to C-style and I-style

The *cli-ignore-modified* statement can be used in: *container*, *list*, *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-incomplete-command

Specifies that an auto-rendered command should be considered incomplete. Can be used to prevent <cr> from appearing in the completion list for optional internal nodes, for example, or to ensure that the user enters all leaf values in a container (if used in combination with cli-sequence-commands).

Used in I- and C-style CLIs.

The *cli-incomplete-command* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:cli-incomplete-no

Specifies that an auto-rendered 'no'-command should not be considered complete, ie, additional leaves or containers must be entered on the same command line.

Used in I- and C-style CLIs.

The *cli-incomplete-no* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-incomplete-show-path

Specifies that a path to the show command is considered incomplete, i.e., it needs more elements added to the path. It can also be used to specify a minimum number of keys to be given for lists.

Used in J-, I- and C-style CLIs.

The *cli-incomplete-show-path* statement can be used in: *leaf*, *tailf:symlink*, *leaf-list*, *list*, *container*, and *refine*.

The following substatements can be used:

tailf:cli-min-keys Specifies the minimum number of required keys for the show command.

tailf:cli-instance-info-leafs *value*

This statement is used to specifiy how list entries are displayed when doing completion in the CLI. By default, a list entry is displayed by listing its key values, and the value of a leaf called 'description', if such a leaf exists in the list entry.

The 'cli-instance-info-leafs' statement takes as its argument a space separated string of leaf names. When a list entry is displayed, the values of these leafs are concatenated with a space character as separator and shown to the user.

For example, when asked to specify an interface the CLI will display a list of possible interface instances, say 1 2 3 4. If the cli-instance-info-leafs property is set to 'description' then the CLI might show:

Possible completions: 1 - internet 2 - lab 3 - dmz 4 - wlan

Used in J-, I- and C-style CLIs.

The *cli-instance-info-leafs* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-key-format *value*

The format string is used when parsing a key value and when generating a key value for an existing configuration. The key items are numbered from 1-N and the format string should indicate how they are related by using $(X) (where X is the key number). For example:

tailf:cli-key-format '$(1)-$(2)' means that the first key item is concatenated with the second key item by a '-'.

Used in J-, I- and C-style CLIs.

The *cli-key-format* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-list-syntax

Specifies that each entry in a leaf-list should be displayed as a separate element.

Used in J-, I- and C-style CLIs.

The *cli-list-syntax* statement can be used in: *leaf-list*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-multi-word Specifies that a multi-word value may be entered without quotes.

tailf:cli-min-column-width *value*

Set a minimum width for the column in the auto-rendered tables.

Used in J-, I- and C-style CLIs.

The *cli-min-column-width* statement can be used in: *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-mode-name *value*

Specifies a custom mode name, instead of the default which is the name of the list or container node.

Can be used in config nodes only. If used in a container, the container must also have a tailf:cli-add-mode statement, and if used in a list, the list must not also have a tailf:cli-suppress-mode statement.

Variables for the list keys in the current mode are available. For examples, 'config-foo-xx$(name)' (privided the key leaf is called 'name').

Used in I- and C-style CLIs.

The *cli-mode-name* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-mode-name-actionpoint *value*

Specifies that a custom function will be invoked to find out the mode name, instead of using the default with is the name of the list or container node.

The argument is the name of an actionpoint, which must be implemented by custom code. In the actionpoint, the command() callback function will be invoked, and it must return a string with the mode name. See confd_lib_dp(3) for details.

Can be used in config nodes only. If used in a container, the container must also have a tailf:cli-add-mode statement, and if used in a list, the list must not also have a tailf:cli-suppress-mode statement.

Used in I- and C-style CLIs.

The *cli-mode-name-actionpoint* statement can be used in: *container*, *list*, and *refine*.

tailf:cli-mount-point *value*

By default actions are mounted under the 'request' command in the J-style CLI and at the top-level in the I- and C-style CLIs. This annotation allowes the action to be mounted under other top level commands

The *cli-mount-point* statement can be used in: *tailf:action* and *rpc*.

tailf:cli-multi-line-prompt

Tells the CLI to automatically enter multi-line mode when prompting the user for a value to this leaf.

Used in J-, I- and C-style CLIs.

The *cli-multi-line-prompt* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-multi-value

Specifies that all remaining tokens on the command line should be considered a value for this leaf. This prevents the need for quoting values containing spaces, but also prevents multiple leaves from being set on the same command line once a multi-value leaf has been given on a line.

If the tailf:cli-max-words substatements is used then additional leaves may be entered.

Used in I- and C-style CLIs.

The *cli-multi-value* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-max-words Specifies the maximum number of allowed words for the key or value.

tailf:cli-multi-word-key

Specifies that the key should allow multiple tokens for the value. Proper type restrictions needs to be used to limit the range of the leaf value.

Can be used in key leafs only.

Used in J-, I- and C-style CLIs.

The *cli-multi-word-key* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-max-words Specifies the maximum number of allowed words for the key or value.

tailf:cli-no-key-completion

Specifies that the CLI engine should not perform completion for key leafs in the list. This is to avoid querying the data provider for all existing keys.

Used in J-, I- and C-style CLIs.

The *cli-no-key-completion* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-no-keyword

Specifies that the name of a node is not present in the CLI.

Note that is must be used with some care, just like tailf:cli-drop-node-name. The resulting data model must still be possible to parse deterministically. For example, consider the data model

```
container interfaces {
   list traffic {
       tailf:cli-no-keyword;
       key id;
       leaf id { type string; }
       leaf mtu { type uint16; }
   }
   list management {
       tailf:cli-no-keyword;
       key id;
       leaf id { type string; }
       leaf mtu { type uint16; }
   }
}
```

In this case it is impossible to determine if the config

```
interfaces {
   eth0 {
      mtu 1400;
    }
}
```

Means that there should be an traffic interface instance named 'eth0' or a management interface instance maned 'eth0'. If, on the other hand, a restriction on the type was used, for example

```
container interfaces {
```

```
   list traffic {
       tailf:cli-no-keyword;
       key id;
       leaf id { type string; pattern 'eth.*'; }
       leaf mtu { type uint16; }
   }
   list management {
       tailf:cli-no-keyword;
       key id;
       leaf id { type string; pattern 'lo.*';}
       leaf mtu { type uint16; }
   }
}
```

then the problem would disappear.

Used in the J-style CLIs.

The *cli-no-keyword* statement can be used in: *leaf*, *container*, *list*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-no-match-completion

Specifies that the CLI engine should not provide match completion for the key leafs in the list.

Used in J-, I- and C-style CLIs.

The *cli-no-match-completion* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-no-name-on-delete

When displaying the deleted version of this element do not include the name.

Applies to C-style

The *cli-no-name-on-delete* statement can be used in: *leaf*, *container*, *list*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-no-value-on-delete

When displaying the deleted version of this leaf do not include the old value.

Applies to C-style

The *cli-no-value-on-delete* statement can be used in: *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-oper-info *text*

This statement works exactly as tailf:info, with the exception that it is used when displaying the element info in the context of stats.

Both tailf:info and tailf:cli-oper-info can be present at the same time.

The *cli-oper-info* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *rpc*, *identity*, *tailf:action*, *tailf:symlink*, and *refine*.

tailf:cli-operational-mode

An action or rpc with this attribute will be available in operational mode, but not in configure mode.

The default is that the action or rpc is available in both configure and operational mode.

Used in J-, I- and C-style CLIs.

The *cli-operational-mode* statement can be used in: *tailf:action* and *rpc*.

tailf:cli-optional-in-sequence

Specifies that this element is optional in the sequence. If it is set it must be set in the right sequence but may be skipped.

Used in I- and C-style CLIs.

The *cli-optional-in-sequence* statement can be used in: *leaf*, *tailf:symlink*, *leaf-list*, *list*, *container*, and *refine*.

tailf:cli-prefix-key

This leaf has to be given as a prefix before entering the actual list keys. Very backwards but a construct that exists in some Cisco CLIs.

The construct can be used also for leaf-lists but only when then tailf:cli-range-list-syntax is also used.

Used in I- and C-style CLIs.

The *cli-prefix-key* statement can be used in: *leaf*, *tailf:symlink*, *refine*, and *leaf-list*.

The following substatements can be used:

tailf:cli-before-key Specifies before which key the prefix element should be inserted. The first key has number 1.

tailf:cli-preformatted

Suppresses quoting of non-config elements when displaying them. Newlines will be preserved in strings etc.

Used in J-, I- and C-style CLIs.

The *cli-preformatted* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-range-delimiters *value*

Allows for custom delimiters to be defined for range expressions. By default only / is considered a delimiter, ie when processing a key like 1/2/3 then each of 1, 2 and 3 will be matched separately agains range expressions, ie given the expression 1-3/5-6/7,8 1 will be matched with 1-3, 2 with 5-6, and 3 with 7,8. If, for example, the delimiters value is set to '/.' then both '/' and '.' will be considered delimiters and an key such as 1/2/3.4 will consist of the enteties 1,2,3,4, all matched separately.

Used in J-, I- and C-style CLIs.

The *cli-range-delimiters* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-range-list-syntax

Specifies that elements in a leaf-list or a list should be entered without surrounding brackets and presented as ranges. The element in the list should be separated by a comma. For example:

vlan 1,3,10-20,30,32,300-310

When this statement is used for lists, the list must have a single key. The elements are be presented as ranges as above.

The type of the list key, or the leaf-list, must be integer based.

Used in J-, I- and C-style CLIs.

The *cli-range-list-syntax* statement can be used in: *leaf-list*, *list*, and *refine*.

tailf:cli-recursive-delete

When generating configuration diffs delete all contents of a container or list before deleting the node.

Applies to C-style

The *cli-recursive-delete* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-remove-before-change

Instructs the CLI engine to generate a no-commnd before modifying an existing instance. It only applies when generating diffs, eg 'show configuration' in C-style.

The *cli-remove-before-change* statement can be used in: *leaf-list*, *list*, *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-replace-all

Specifies that the new leaf-list value(s) should replace the old, as opposed to be added to the old leaf-list.

The *cli-replace-all* statement can be used in: *leaf-list*, *tailf:cli-flat-list-syntax*, and *refine*.

tailf:cli-reset-container

Specifies that all sibling leaves in the container should be reset when this element is set.

When used on a container its content is cleared when set.

The *cli-reset-container* statement can be used in: *leaf*, *list*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-run-template *value*

Specifies a template string to be used by the 'show running-config' command in operational mode. It is primarily intended for displaying config data but non-config data may be included in the template as well.

See the defintion of cli-template-string for more info.

Used in I- and C-style CLIs.

The *cli-run-template* statement can be used in: *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-run-template-enter *value*

Specifies a template string to be printed before each list entry is printed.

When used on a container it only has effect when the container also has a tailf:cli-add-mode, and when tailf:cli-show-no isn't used on the container.

See the defintion of cli-template-string for more info.

The variable .reenter is set to 'true' when the 'show configuration' command is executed and the list or container isn't created. This allow, for example, to display

create foo

when an instance is created

edit foo

when something inside the instance is modified.

Used in I- and C-style CLIs.

The *cli-run-template-enter* statement can be used in: *list*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-run-template-footer *value*

Specifies a template string to be printed after all list entries are printed.

See the defintion of cli-template-string for more info.

Used in I- and C-style CLIs.

The *cli-run-template-footer* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-run-template-legend *value*

Specifies a template string to be printed before all list entries are printed.

See the defintion of cli-template-string for more info.

Used in I- and C-style CLIs.

The *cli-run-template-legend* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-sequence-commands

Specifies that an auto-rendered command should only accept arguments in the same order as they are specified in the YANG model. This, in combination with tailf:cli-drop-node-name, can be used to create CLI commands for setting multiple leafs in a container without having to specify the leaf names.

In almost all cases this annotation should be accompanied by the tailf:cli-compact-syntax annotation. Otherwise the output from 'show running-config' will not be correct, and the sequence 'save xx' 'load override xx' will not work.

Used in I- and C-style CLIs.

The *cli-sequence-commands* statement can be used in: *list*, *container*, *tailf:symlink*, and *refine*.

The following substatements can be used:

tailf:cli-reset-siblings Specifies that all sibling leaves in the sequence should be reset whenever the first leaf in the sequence is set.

tailf:cli-reset-all-siblings Specifies that all sibling leaves in the container should be reset whenever the first leaf in the sequence is set.

tailf:cli-show-config

Specifies that the node will be included when doing a 'show running-configuration', even if it is a non-config node.

Used in I- and C-style CLIs.

The *cli-show-config* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *refine*, and *tailf:symlink*.

tailf:cli-show-long-obu-diffs

Instructs the CLI engine to not generate 'insert' comments when displaying configuration changes of ordered-by user lists, but instead explicitly remove old instances with 'no' and then add the instances following a newly inserted instance. Should not be used together with tailf:cli-show-obu-comments

The *cli-show-long-obu-diffs* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-show-no

Specifies that an optional leaf node or presence container should be displayed as 'no <name>' when it does not exist. For example, if a leaf 'shutdown' has this property and does not exist, 'no shutdown' is displayed.

Used in I- and C-style CLIs.

The *cli-show-no* statement can be used in: *leaf*, *list*, *leaf-list*, *refine*, *tailf:symlink*, and *container*.

tailf:cli-show-obu-comments

Enforces the CLI engine to generate 'insert' comments when displaying configuration changes of ordered-by user lists. Should not be used together with tailf:cli-show-long-obu-diffs

The *cli-show-obu-comments* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-show-order-tag *value*

Specifies a custom display order for nodes with the tailf:cli-show-order-tag attribute. Nodes will be displayed in the order indicated by a cli-show-order-taglist attribute in a parent node.

The scope of a tag reaches until a new taglist is encountered.

Used in I- and C-style CLIs.

The *cli-show-order-tag* statement can be used in: *container*, *list*, *leaf*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-show-order-taglist *value*

Specifies a custom display order for nodes with the tailf:cli-show-order-tag attribute. Nodes will be displayed in the order indicated in the list. Nodes without a tag will be displayed after all nodes with a tag have been displayed.

The scope of a taglist is until a new taglist is encountered.

Used in I- and C-style CLIs.

The *cli-show-order-taglist* statement can be used in: *container*, *list*, *tailf:symlink*, and *refine*.

tailf:cli-show-template *value*

Specifies a template string to be used by the 'show' command in operational mode. It is primarily intended for displaying non-config data but config data may be included in the template as well.

See the defintion of cli-template-string for more info.

Used in J-, I- and C-style CLIs.

The *cli-show-template* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

The following substatements can be used:

tailf:cli-auto-legend Specifies that the legend should be automatically rendered if not already displayed. Useful when using templates for rendering tables.

tailf:cli-show-template-enter *value*

Specifies a template string to be printed before each list entry is printed.

See the defintion of cli-template-string for more info.

Used in J-, I- and C-style CLIs.

The *cli-show-template-enter* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-show-template-footer *value*

Specifies a template string to be printed after all list entries are printed.

See the defintion of cli-template-string for more info.

Used in J-, I- and C-style CLIs.

The *cli-show-template-footer* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-show-template-legend *value*

Specifies a template string to be printed before all list entries are printed.

See the defintion of cli-template-string for more info.

Used in J-, I- and C-style CLIs.

The *cli-show-template-legend* statement can be used in: *list* and *refine*.

tailf:cli-show-with-default

This leaf will be displayed even when it has its default value. Note that this will somewhat result in a slightly different behaviour when you save a config and then load it again. With this setting in place a leaf that has not been configured will be configured after the load.

Used in I- and C-style CLIs.

The *cli-show-with-default* statement can be used in: *leaf*, *refine*, and *tailf:symlink*.

tailf:cli-strict-leafref

Specifies that the leaf should only be allowed to be assigned references to existing instances when the command is executed. Without this annotation the requirement is that the instance exists on commit time.

Used in I- and C-style CLIs.

The *cli-strict-leafref* statement can be used in: *leaf, leaf-list, tailf:symlink,* and *refine*.

tailf:cli-suppress-key-abbreviation

Key values cannot be abbreviated. The user must always give complete values for keys.

In the J-style CLI this is relevant when using the commands 'delete' and 'edit'.

In the I- and C-style CLIs this is relevant when using the commands 'no', 'show configuration' and for commands to enter submodes.

See also /confdConfig/cli/allowAbbrevKeys in confd.conf(5).

The *cli-suppress-key-abbreviation* statement can be used in: *list, tailf:symlink,* and *refine*.

tailf:cli-suppress-key-sort

Instructs the CLI engine to not sort the keys in alphabetical order when presenting them to the user during TAB completion.

Used in J-, I- and C-style CLIs.

The *cli-suppress-key-sort* statement can be used in: *list, tailf:symlink,* and *refine*.

tailf:cli-suppress-list-no

Specifies that the CLI should not accept deletion of the entire list or leaf-list. Only specific instances should be deletable not the entire list in one command. ie, 'no foo <instance>' should be allowed but not 'no foo'.

Used in I- and C-style CLIs.

The *cli-suppress-list-no* statement can be used in: *leaf-list, list, tailf:symlink,* and *refine*.

tailf:cli-suppress-mode

Instructs the CLI engine to not make a mode of the list node.

Can be used in config nodes only.

Used in I- and C-style CLIs.

The *cli-suppress-mode* statement can be used in: *list, tailf:symlink,* and *refine*.

tailf:cli-suppress-no

Specifies that the CLI should not auto-render 'no' commands for this element. An element with this annotation will not appear in the completion list to the 'no' command.

Used in I- and C-style CLIs.

The *cli-suppress-no* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-suppress-range

Means that the integer key should not allow range expressions.

Can be used in key leafs only.

Used in J-, I- and C-style CLIs.

The *cli-suppress-range* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-suppress-shortenabled

Suppresses the confd.conf(5) setting /confdConfig/cli/useShortEnabled.

Used in J-, I- and C-style CLIs.

The *cli-suppress-shortenabled* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-suppress-show-conf-path

Specifies that the show running-config command cannot be invoked with the path, ie the path is suppressed when auto-rendering show running- config commands for config='true' data.

Used in J-, I- and C-style CLIs.

The *cli-suppress-show-conf-path* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, and *refine*.

tailf:cli-suppress-show-match

Specifies that a specific completion match (i.e., a filter match that appear at list nodes as an alternative to specifying a single instance) to the show command should not be available.

Used in J-, I- and C-style CLIs.

The *cli-suppress-show-match* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-suppress-show-path

Specifies that the show command cannot be invoked with the path, ie the path is suppressed when auto-rendering show commands for config='false' data.

Used in J-, I- and C-style CLIs.

The *cli-suppress-show-path* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-suppress-silent-no *value*

Specifies that the confd.cnof directive cSilentNo should be suppressed for a leaf and that a custom error message should be displayed when the user attempts to delete a non-existing element.

Used in I- and C-style CLIs.

The *cli-suppress-silent-no* statement can be used in: *leaf*, *leaf-list*, *list*, *tailf:symlink*, *container*, and *refine*.

tailf:cli-suppress-table

Instructs the CLI engine to not print the list as a table in the 'show' command.

Can be used in non-config nodes only.

Used in I- and C-style CLIs.

The *cli-suppress-table* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-suppress-validation-warning-prompt

Instructs the CLI engine to not prompt the user whether to proceed or not if a warning is generated for this node.

Used in I- and C-style CLIs.

The *cli-suppress-validation-warning-prompt* statement can be used in: *list*, *leaf*, *container*, *leaf-list*, *tailf:symlink*, and *refine*.

tailf:cli-suppress-wildcard

Means that the list does not allow wildcard expressions in the 'show' pattern.

See also /confdConfig/cli/allowWildcard in confd.conf(5).

Used in J-, I- and C-style CLIs.

The *cli-suppress-wildcard* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-table-footer *value*

Specifies a template string to be printed after all list entries are printed.

Used in J-, I- and C-style CLIs.

The *cli-table-footer* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-table-legend *value*

Specifies a template string to be printed before all list entries are printed.

Used in J-, I- and C-style CLIs.

The *cli-table-legend* statement can be used in: *list*, *tailf:symlink*, and *refine*.

tailf:cli-trim-default

Do not display value if it is same as default.

Used in I- and C-style CLIs.

The *cli-trim-default* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

tailf:cli-value-display-template *value*

Specifies a template string to be used when formating the value of a leaf for display. Note that other leaves cannot be referenced from a display template of one leaf. The only value accessible is the leaf's own value, accessed through $(.).

See the defintion of cli-template-string for more info.

Used in J-, I- and C-style CLIs.

The *cli-value-display-template* statement can be used in: *leaf*, *tailf:symlink*, and *refine*.

YANG TYPES

cli-template-string

A template is a text string which is expanded by the CLI engine, and then displayed to the user.

The template may contain a mix of text and expandable entries. Expandable entries all start with $(and end with a matching). Parentheses and dollar signs need to be quoted in plain text.

The template is expanded as follows:

A parameter is either a relative or absolute path to a leaf element (eg /foo/bar, foo/bar), or one of the builtin variables: .selected, .entered, .legend_shown, .user, .groups, .ip, .display_groups, .path, .ipath or .licounter. In addition the variables .spath and .ispath are available when a command is executed from a show path.

.selected

The .selected variable contains the list of selected paths to be shown. The show template can inspect this element to determine if a given element should be displayed or not. For example:

$(.selected~=hwaddr?HW Address)

.entered

The .entered variable is true if the "entered" text has been displayed (either the auto generated text or a showTemplateEnter). This is useful when having a non-table template where each instance should have a text.

$(.entered?:host $(name))

.legend_shown

The .legend_shown variable is true if the "legend" text has been displayed (either the auto generated table header or a showTemplateLegend). This is useful to inspect when displaying a table row. If the user enteres the path to a specific instance the builtin table header will not be displayed and the showTemplateLegend will not be invoked and it may be useful to render the legend specifically for this instance.

$(.legend_shown!=true?Address Interface)

.user

The .user variable contains the name of the current user. This can be used for differentiating the content displayed for a specific user, or in paths. For exapmle:

```
        $(user{$(.user)}/settings)
```

.groups

The .groups variable contains the a list of groups that the user belongs to.

.display_groups

The .display_groups variable contains a list of selected display groups. This can be used to display different content depending on the selected display group. For example:

$(.display_groups~=details?details...)

.ip

The .ip variable contains the ip address that the user connected from.

.path

The .path variable contains the path to the entry, formated in CLI style.

.ipath

The .ipath variable contains the path to the entry, formated in template style.

.spath

The .spath variable contains the show path, formated in CLI style.

.ispath

The .ispath variable contains the show path, formated in template style.

.licounter

The .licounter variable contains a counter that is incremented for each instance in a list. This means that it will be 0 in the legend, contain the total number of list instances in the footer and something in between in the basic show template.

$(parameter)

The value of 'parameter' is substituted.

$(cond?word1:word2)

The expansion of 'word1' is substituted if 'cond' evaluates to true, otherwise the expansion of 'word2' is substituted.

'cond' may be one of

parameter

Evaluates to true if the node exists.

parameter == <value>

Evaluates to true if the value of the parameter equals <value>.

parameter != <value>

Evalutes to true if the value of the parameter does not equal <value>

parameter ~= <value>

Provided that the value of the parameter is a list (i.e., the node that the parameter refers to is a leaf-list), this expression evaluates to true if <value> is a member of the list.

$(parameter|filter)

The value of 'parameter' processed by 'filter' is substituted. Filters may be either one of the built-ins or a customized filter defined in a callback. See /confdConfig/cli/templateFilter.

A built-in 'filter' may be one of:

capfirst

Capitalizes the first character of the value.

lower

Converts the value into lowercase.

upper

Converts the value into uppercase.

filesizeformat

Formats the value in a human-readable format (e.g., '13 KB', '4.10 MB', '102 bytes' etc), where K means 1024, M means 1024*1024 etc.

When used without argument the default number of decimals displayed is 2. When used with a numeric integer argument, filesizeformat will display the given number of decimal places.

humanreadable

Similar to filesizeformat except no bytes suffix is added (e.g., '13.00 k', '4.10 M' '102' etc), where k means 1000, M means 1000*1000 etc.

When used without argument the default number of decimals displayed is 2. When used with a numeric integer argument, humanreadable will display the given number of decimal places.

commasep

Separate the numerical values into groups of three digits using a comma (e.g., 1234567 -> 1,234,567)

hex

Display integer as hex number. An argument can be used to indicate how many digits should be used in the output. If the hex number is too long it will be truncated at the front, if it is too short it will be padded with zeros at the front. If the width is a negative number then at most that number of digits will be used, but short numbers will not be padded with zeroes. Another argument can be given to indicate if the hex numbers should be written with lower or upper case.

For example:

```
value           Template                    Output
12345          {{ value|hex }}              3039
12345          {{ value|hex:2 }}            39
12345          {{ value|hex:8 }}            00003039
```

```
12345           {{ value|hex:-8 }}              3039
14911           {{ value|hex:-8:upper }}        3A3F
14911           {{ value|hex:-8:lower }}        3a3f
```

hexlist

Display integer as hex number with : between pairs. An argument can be used to indicate how many digits should be used in the output. If the hex number is too long it will be truncated at the front, if it is too short it will be padded with zeros at the front. If the width is a negative number then at most that number of digits will be used, but short numbers will not be padded with zeroes. Another argument can be given to indicate if the hex numbers should be written with lower or upper case.

For example:

```
value            Template                       Output
12345           {{ value|hexlist }}             30:39
12345           {{ value|hexlist:2 }}           39
12345           {{ value|hexlist:8 }}           00:00:30:39
12345           {{ value|hexlist:-8 }}          30:39
14911           {{ value|hexlist:-8:upper }}    3A:3F
14911           {{ value|hexlist:-8:lower }}    3a:3f
```

floatformat

Used for type 'float' in tailf-xsd-types. We recommend that the YANG built-in type 'decimal64' is used instead of 'float'.

When used without an argument, rounds a floating-point number to one decimal place -- but only if there is a decimal part to be displayed.

For example:

```
value           Template                        Output
34.23234        {{ value|floatformat }}         34.2
34.00000        {{ value|floatformat }}         34
34.26000        {{ value|floatformat }}         34.3
```

If used with a numeric integer argument, floatformat rounds a number to that many decimal places. For example:

```
value           Template                        Output
34.23234        {{ value|floatformat:3 }}       34.232
34.00000        {{ value|floatformat:3 }}       34.000
34.26000        {{ value|floatformat:3 }}       34.260
```

If the argument passed to floatformat is negative, it will round a number to that many decimal places -- but only if there's a decimal part to be displayed. For example:

```
value           Template                        Output
34.23234        {{ value|floatformat:-3 }}      34.232
34.00000        {{ value|floatformat:-3 }}      34
34.26000        {{ value|floatformat:-3 }}      34.260
```

Using floatformat with no argument is equivalent to using floatformat with an argument of -1.

ljust:width

Left-align the value given a width.

rjust:width

Right-align the value given a width.

trunc:width

Truncate value to a given width.

lower

Convert the value into lowercase.

upper

Convert the value into uppercase.

show:<dictionary>

Substitutes the result of invoking the default display function for the parameter. The dictionary can be used for introducing own variables that can be accessed in the same manner as builtin variables. The user defined variables overrides builtin variables. The dictionary is specified as a string on the following form:

(key=value)(:key=value)*

For example, with the following expression:

$(foo|show:myvar1=true:myvar2=Interface)

the user defined variables can be accessed like this:

$(.myvar1!=true?Address) $(.myvar2)

A special case is the dict variable 'indent'. It controls the indentation level of the displayed path. The current indent level can be incremented and decremented using =+ and =-.

For example:

$(foobar|show:indent=+2) $(foobar|show:indent=-1) $(foobar|show:indent=10)

Another special case is he dict variable 'noalign'. It may be used to suppress the default aligning that may occur when displaying an element.

For example:

$(foobar|show:noalign)

dict:<dictionary>

Translates the value using the dictionary. Can for example be used for displaying on/off instead of true/false. The dictionary is specified as a string on the following form:

(key=value)(:key=value)*

For example, with the following expression:

$(foo|dict:true=on:false=off)

if the leaf 'foo' has value 'true', it is displayed as 'on', and if its value is 'false' it is displayed as 'off'.

```
Nested invocations are allowed, ie it is possible to have expressions
like $((state|dict:yes=Yes:no=No)|rjust:14), or $(/foo{$(../bar)})
```

For example:

```
list interface {
  key name;
  leaf name { ... }
  leaf status { ... }
  container line {
    leaf status { ... }
  }
  leaf mtu { ... }
  leaf bw { ... }
  leaf encapsulation { ... }
  leaf loopback { ... }
  tailf:cli-show-template
    '$(name) is administratively $(status),'
  + ' line protocol is $(line/status)\n'
  + 'MTU $(mtu) bytes, BW $(bw|humanreadable)bit, \n'
  + 'Encap $(encapsulation|upper), $(loopback?:loopback not set)\n';
}
```

SEE ALSO

The User Guide

confdc(1)	ConfD compiler
tailf_yang_extensions(5)	Tail-f YANG extensions

Name

tailf_yang_extensions — Tail-f YANG extensions

Synopsis

```
tailf:action

tailf:actionpoint

tailf:alt-name

tailf:annotate

tailf:annotate-module

tailf:callpoint

tailf:cdb-oper

tailf:code-name

tailf:confirm-text

tailf:default-ref

tailf:dependency

tailf:display-column-name

tailf:display-groups

tailf:display-hint

tailf:display-status-name

tailf:display-when

tailf:error-info

tailf:exec

tailf:export

tailf:hidden

tailf:id

tailf:id-value

tailf:indexed-view

tailf:info

tailf:info-html

tailf:java-class-name
```

```
tailf:junos-val-as-xml-tag
tailf:junos-val-with-prev-xml-tag
tailf:key-default
tailf:link
tailf:lower-case
tailf:ncs-device-type
tailf:no-dependency
tailf:non-strict-leafref
tailf:path-filters
tailf:secondary-index
tailf:snmp-delete-value
tailf:snmp-exclude-object
tailf:snmp-lax-type-check
tailf:snmp-mib-module-name
tailf:snmp-name
tailf:snmp-ned-accessible-column
tailf:snmp-ned-delete-before-create
tailf:snmp-ned-modification-dependent
tailf:snmp-ned-recreate-when-modified
tailf:snmp-ned-set-before-row-modification
tailf:snmp-oid
tailf:snmp-row-status-column
tailf:sort-order
tailf:sort-priority
tailf:step
tailf:structure
tailf:suppress-echo
tailf:symlink
tailf:typepoint
tailf:unique-selector
```

`tailf:validate`

`tailf:value-length`

`tailf:writable`

`tailf:xpath-root`

DESCRIPTION

This manpage describes all the Tail-f extensions to YANG. The YANG extensions consist of YANG statements and XPath functions to be used in YANG data models.

The YANG source file `$CONFD_DIR/src/confd/yang/tailf-common.yang` gives the exact YANG syntax for all Tail-f YANG extension statements - using the YANG language itself.

Most of the concepts implemented by the extensions listed below are described in the ConfD User Guide. For example user defined validation is described in the Validation chapter. The YANG syntax is described here though.

YANG STATEMENTS

tailf:action *name*

Defines an action (method) in the data model.

When the action is invoked, the instance on which the action is invoked is explicitly identified by an hierarchy of configuration or state data.

The action statement can have either a 'tailf:actionpoint' or a 'tailf:exec' substatement. If the action is implemented as a callback in an application daemon, 'tailf:actionpoint' is used, whereas 'tailf:exec' is used for an action implemented as a standalone executable (program or script). Additionally, 'action' can have the same substatements as the standard YANG 'rpc' statement, e.g., 'description', 'input', and 'output'.

For example:

```
container sys {
  list interface {
    key name;
    leaf name {
      type string;
    }
    tailf:action reset {
      tailf:actionpoint my-ap;
      input {
        leaf after-seconds {
          mandatory false;
          type int32;
        }
      }
    }
  }
}
```

We can also add a 'tailf:confirm-text', which defines a string to be used in the user interfaces to prompt the user for confirmation before the action is executed. The optional 'tailf:confirm-default' and 'tailf:cli-

batch-confirm-default' can be set to control if the default is to proceed or to abort. The latter will only be used during batch processing in the CLI (e.g. non-interactive mode).

```
tailf:action reset {
  tailf:actionpoint my-ap;
  input {
    leaf after-seconds {
      mandatory false;
      type int32;
    }
  }
  tailf:confirm-text 'Really want to do this?' {
    tailf:confirm-default true;
  }
}
```

The 'tailf:actionpoint' statement can have a 'tailf:opaque' substatement, to define an opaque string that is passed to the callback function.

```
tailf:action reset {
  tailf:actionpoint my-ap {
    tailf:opaque 'reset-interface';
  }
  input {
    leaf after-seconds {
      mandatory false;
      type int32;
    }
  }
}
```

When we use the 'tailf:exec' substatement, the argument to exec specifies the program or script that should be executed. For example:

```
tailf:action reboot {
  tailf:exec '/opt/sys/reboot.sh' {
    tailf:args '-c $(context) -p $(path)';
  }
  input {
    leaf when {
      type enumeration {
        enum now;
        enum 10secs;
        enum 1min;
      }
    }
  }
}
```

The *action* statement can be used in: *augment*, *list*, *container*, and *grouping*.

The following substatements can be used:

tailf:actionpoint

tailf:alt-name

tailf:cli-mount-point

tailf:cli-configure-mode

tailf:cli-operational-mode

tailf:cli-oper-info

tailf:code-name

tailf:confirm-text

tailf:display-when

tailf:exec

tailf:hidden

tailf:info

tailf:info-html

tailf:actionpoint *name*

Identifies the callback in a data provider that implements the action. See confd_lib_dp(3) for details on the API.

The *actionpoint* statement can be used in: *rpc*, *tailf:action*, and *refine*.

The following substatements can be used:

tailf:opaque Defines an opaque string which is passed to the callback function in the context.

tailf:internal For internal ConfD / NCS use only.

tailf:alt-name *name*

This property is used to specify an alternative name for the node in the CLI. It is used instead of the node name in the CLI, both for input and output.

The *alt-name* statement can be used in: *rpc*, *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:annotate *target*

Annotates an existing statement with a 'tailf' statement or a validation statement. This is useful in order to add tailf statements to a module without touching the module source. Annotation statements can be put in a separate annotation module, and then passed to 'confdc' (or 'pyang') when the original module is compiled.

Any 'tailf' statement, except 'symlink' and 'action' can be annotated. The statements 'symlink' and 'action' modifies the data model, and are thus not allowed.

The validation statements 'must', 'min-elements', 'max-elements', 'mandatory', 'unique', and 'when' can also be annotated.

A 'description' can also be annotated.

'tailf:annotate' can occur on the top-level in a module, or in another 'tailf:annotate' statement.

The argument is a 'schema-nodeid', i.e. the same as for 'augment', or a '*'. It identifies a target node in the schema tree to annotate with new statements. The special value '*' can be used within another 'tailf:annotate' statetement, to select all children for annotation.

The target node is searched for after 'uses' and 'augment' expansion. All substatements to 'tailf:annotate' are treated as if they were written inline in the target node, with the exception of any 'tailf:annotate' substatements. These are treated recursively. For example, the following snippet adds one callpoint to /x and one to /x/y:

```
tailf:annotate /x {
  tailf:callpoint xcp;
  tailf:annotate y {
    tailf:callpoint ycp;
  }
}
```

The *annotate* statement can be used in: *module* and *submodule*.

The following substatements can be used:

tailf:annotate

tailf:annotate-module *module-name*

Annotates an existing module or submodule statement with a 'tailf' statement. This is useful in order to add tailf statements to a module without touching the module source. Annotation statements can be put in a separate annotation module, and then passed to 'confdc' (or 'pyang') when the original module is compiled.

'tailf:annotate-module' can occur on the top-level in a module, and is used to add 'tailf' statements to the module statement itself.

The argument is a name of the module or submodule to annotate.

The *annotate-module* statement can be used in: *module*.

The following substatements can be used:

tailf:snmp-oid

tailf:snmp-mib-module-name

tailf:id

tailf:id-value

tailf:export

tailf:unique-selector

tailf:annotate-statement Annotates an existing statement with a 'tailf' statement, a validation statement, or a type restrcition statement. This is useful in order to add tailf statements to a module without touching the module source. Annotation statements can be put in a separate annotation module, and then passed to 'confdc' (or 'pyang') when the original module is compiled.

Any 'tailf' statement, except 'symlink' and 'action' can be annotated. The statements 'symlink' and 'action' modifies the data model, and are thus not allowed.

The validation statements 'must', 'min-elements', 'max-elements', 'mandatory', 'unique', and 'when' can also be annotated.

The type restriction statement 'pattern' can also be annotated.

A 'description' can also be annotated.

The argument is an XPath-like expression that selects a statement to annotate. The syntax is:

<statement-name> ('[' <arg-name> '=' <arg-value> ']')

where <statement-name> is the name of the statement to annotate, and if there are more than one such statement in the parent, <arg-value> is the quoted value of the statement's argument.

All substatements to 'tailf:annotate-statement' are treated as if they were written inline in the target node, with the exception of any 'tailf:annotate-statement' substatements. These are treated recursively.

For example, given the grouping:

grouping foo { leaf bar { type string; } leaf baz { type string; } }

the following snippet adds a callpoint to the leaf 'baz':

tailf:annotate-statement grouping[name='foo'] { tailf:annotate-statement leaf[name='baz'] { tailf:callpoint xcp; } }

tailf:callpoint *id*

Identifies a callback in a data provider. A data provider implements access to external data, either configuration data in a database or operational data. By default ConfD uses the embedded database (CDB) to store all data. However, some or all of the configuration data may be stored in an external source. In order for ConfD to be able to manipulate external data, a data provider registers itself using the callpoint id as described in confd_lib_dp(3).

A callpoint is inherited to all child nodes unless another 'callpoint' or an 'cdb-oper' is defined.

The *callpoint* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *refine*, and *grouping*.

The following substatements can be used:

tailf:config If this statement is present, the callpoint is applied to nodes with a matching value of their 'config' property.

tailf:transform If set to 'true', the callpoint is a transformation callpoint. How transformation callpoints are used is described in the 'Transformations, Hooks, Hidden Data and Symlinks' chapter in the User's Guide.

tailf:set-hook Set hooks are a means to associate user code to the transaction. Whenever an element gets written, created, or deleted, user code gets invoked and can optionally write more data into the same transaction.

The difference between set- and transaction hooks are that set hooks are invoked immediately when an element is modified, but transaction hooks are invoked at commit time.

The value 'subtree' means that all nodes in the configuration below where the hook is defined are affected.

The value 'object' means that the hook only applies to the list where it is defined, i.e. it applies to all child nodes that are not themselves lists.

The value 'node' means that the hook only applies to the node where it is defined and none of its children.

For more details on hooks, see the 'Transformations, Hooks, Hidden Data and Symlinks' chapter in the User's Guide.

tailf:transaction-hook Transaction hooks are a means to associate user code to the transaction. Whenever an element gets written, created, or deleted, user code gets invoked and can optionally write more data into the same transaction.

The difference between set- and transaction hooks are that set hooks are invoked immediately when an element is modified, but transaction hooks are invoked at commit time.

The value 'subtree' means that all nodes in the configuration below where the hook is defined are affected.

The value 'object' means that the hook only applies to the list where it is defined, i.e. it applies to all child nodes that are not themselves lists.

The value 'node' means that the hook only applies to the node where it is defined and none of its children.

For more details on hooks, see the 'Transformations, Hooks, Hidden Data and Symlinks' chapter in the User's Guide.

tailf:cache If set to 'true', the operational data served by the callpoint will be cached by ConfD. If set to 'true' in a node that represents configuration data, the statement 'tailf:config' must be present and set to 'false'. This feature is further described in the section 'Caching operational data' in the 'Operational data' chapter in the User's Guide.

tailf:opaque Defines an opaque string which is passed to the callback function in the context.

tailf:internal For internal ConfD / NCS use only.

tailf:cdb-oper

Indicates that operational data nodes below this node are stored in CDB.

The *cdb-oper* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

The following substatements can be used:

tailf:persistent If it is set to 'true', the operational data is stored on disk. If set to 'false', the operational data is not persistent across ConfD restarts. The default is 'false'.

tailf:code-name *name*

Used to give another name to the enum or node name in generated header files. This statement is typically used to avoid name conflicts if there is a data node with the same name as the enumeration, if there are multiple enumerations in different types with the same name but different values, or if there are multiple node names that are mapped to the same name in the header file.

The *code-name* statement can be used in: *enum*, *bit*, *leaf*, *leaf-list*, *list*, *container*, *rpc*, *identity*, *notification*, and *tailf:action*.

tailf:confirm-text *text*

A string which is used in the user interfaces to prompt the user for confirmation before the action is executed. The optional 'confirm-default' and 'cli-batch-confirm-default' can be set to control if the default is to proceed or to abort. The latter will only be used during batch processing in the CLI (e.g. non-interactive mode).

The *confirm-text* statement can be used in: *rpc* and *tailf:action*.

The following substatements can be used:

tailf:confirm-default Specifies if the default is to proceed or abort the action when a confirm-text is set. If this value is not specified, a ConfD global default value can be set in clispec(5).

tailf:cli-batch-confirm-default

tailf:default-ref *path*

This statement defines a dynamic default value. It is a reference to some other leaf in the datamodel. If no value has been set for this leaf, it defaults to the value of the leaf that the 'default-ref' argument points to.

The textual format of a 'default-ref' is an XPath location path with no predicates.

The type of the leaf with a 'default-ref' will be set to the type of the referred leaf. This means that the type statement in the leaf with the 'default-ref' is ignored, but it SHOULD match the type of the referred leaf.

Here is an example, where a group without a 'hold-time' will get as default the value of another leaf up in the hierarchy:

```
leaf hold-time {
    mandatory true;
    type int32;
}
list group {
    key 'name';
    leaf name {
        type string;
    }
    leaf hold-time {
        type int32;
        tailf:default-ref '../../hold-time';
    }
}
```

The *default-ref* statement can be used in: *leaf* and *refine*.

tailf:dependency *path*

This statement is used to specify that the must or when expression or validation function depends on a set of subtrees in the data store. Whenever a node in one of those subtrees are modified, the must or when expression is evaluated, or validation code executed.

The textual format of a 'dependency' is an XPath location path with no predicates.

If the node that declares the dependency is a leaf, there is an implicit dependency to the leaf itself.

For example, with the leafs below, the validation code for'vp' will be called whenever 'a' or 'b' is modified.

```
leaf a {
    type int32;
    tailf:validate vp {
        tailf:dependency '../b';
    }
}
leaf b {
    type int32;
}
```

For 'when' and 'must' expressions, the compiler can derive the dependencies automatically from the XPath expression in most cases. The exception is if any wildcards are used in the expression.

For 'when' expressions to work, a 'tailf:dependency' statement must be given, unless the compiler can figure out the dependency by itself.

Note that having 'must' expressions or a 'tailf:validate' statement without dependencies impacts the overall performance of the system, since all such 'must' expressions or validation functions are evaluated at every commit.

The *dependency* statement can be used in: *must*, *when*, and *tailf:validate*.

The following substatements can be used:

tailf:xpath-root

tailf:display-column-name *name*

This property is used to specify an alternative column name for the leaf in the CLI. It is used when displaying the leaf in a table in the CLI.

The *display-column-name* statement can be used in: *leaf*, *leaf-list*, and *refine*.

tailf:display-groups *value*

This property is used in the CLI when 'enableDisplayGroups' has been set to true in the confd.conf(5) file. Display groups are used to control which elements should be displayed by the show command.

The argument is a space-separated string of tags.

In the J-style CLI the 'show status', 'show table' and 'show all' commands use display groups. In the C- and I-style CLIs the 'show <pattern>' command uses display groups.

If no display groups are specified when running the commands, the node will be displayed if it does not have the 'display-groups' property, or if the property value includes the special value 'none'.

If display groups are specified when running the command, then the node will be displayed only if its 'display-group' property contains one of the specified display groups.

The *display-groups* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:display-hint *hint*

This statement can be used to add a display-hint to a leaf or typedef of type binary. The display-hint is used in the CLI and WebUI instead of displaying the binary as a base64-encoded string. It is also used for input.

The value of a 'display-hint' is defined in RFC 2579.

For example, with the display-hint value '1x:', the value is printed and inputted as a colon-separated hex list.

The *display-hint* statement can be used in: *leaf* and *typedef*.

tailf:display-status-name *name*

This property is used to specify an alternative name for the element in the CLI. It is used when displaying status information in the C- and I-style CLIs.

The *display-status-name* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:display-when *condition*

The argument contains an XPath expression which specifies when the node should be displayed in the CLI and WebUI. For example, when the CLI performs completion, and one of the candidates is a node with a 'display-when' expression, the expression is evaluated by the CLI. If the XPath expression evaluates to true, the node is shown as a possible completion candidate, otherwise not.

For a list, the display-when expression is evaluated once for the entire list. In this case, the XPath context node is the list's parent node.

This feature is further described in the 'Transformations, Hooks, Hidden Data and Symlinks' chapter in the User Guide.

The *display-when* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

The following substatements can be used:

tailf:xpath-root

tailf:error-info

Declares a set of data nodes to be used in the NETCONF <error-info> element.

A data provider can use one of the confd_*_seterr_extended_info() functions (see confd_lib_dp(3)) to set these data nodes on errors.

This statement may be used multiple times.

For example:

```
  tailf:error-info {
     leaf severity {
       type enumeration {
         enum info;
         enum error;
         enum critical;
       }
     }
     container detail {
       leaf class {
         type uint8;
       }
       leaf code {
         type uint8;
       }
     }
   }
```

The *error-info* statement can be used in: *module* and *submodule*.

tailf:exec *cmd*

Specifies that the rpc or action is implemented as an OS executable. The argument 'cmd' is the path to the executable file. If the command is in the $PATH of ConfD, the 'cmd' can be just the name of the executable.

The *exec* statement can be used in: *rpc* and *tailf:action*.

The following substatements can be used:

tailf:args Specifies arguments to send to the executable when it is invoked by ConfD. The argument 'value' is a space separated list of argument strings. It may contain variables on the form $(variablename). These variables will be expanded before the command is executed. The following variables are always available:

$(user) The name of the user which runs the operation.

$(groups) A comma separated string of the names of the groups the user belongs to.

$(ip) The source ip address of the user session.

$(uid) The user id of the user.

$(gid) The group id of the user.

When the parent 'exec' statement is a substatement of 'action', the following additional variablenames are available:

$(keypath) The path that identifies the parent container of 'action' in string keypath form, e.g., '/sys:host{earth}/interface{eth0}'.

$(path) The path that identifies the parent container of 'action' in CLI path form, e.g., 'host earth interface eth0'.

$(context) cli | webui | netconf | any string provided by MAAPI

For example: args '-user $(user) $(uid)'; might expand to: -user bob 500

tailf:uid Specifies which user id to use when executing the command.

If 'uid' is an integer value, the command is run as the user with this user id.

If 'uid' is set to either 'user', 'root' or an integer user id, the ConfD daemon must have been started as root (or setuid), or the ConfD executable program 'cmdwrapper' must have setuid root permissions.

tailf:gid Specifies which group id to use when executing the command.

If 'gid' is an integer value, the command is run as the group with this group id.

If 'gid' is set to either 'user', 'root' or an integer group id, the ConfD daemon must have been started as root (or setuid), or the ConfD executable program 'cmdwrapper' must have setuid root permissions.

tailf:wd Specifies which working directory to use when executing the command. If not given the command is executed from the homedir of the user logged in to ConfD.

tailf:global-no-duplicate Specifies that only one instance with the same name can be run at any one time in the system. The command can be started either from the CLI, the WebUI or through NETCONF. If a client tries to execute this command while another operation with the same 'global-no-duplicate' name is running, a 'resource-denied' error is generated.

tailf:raw-xml Specifies that ConfD should not convert the RPC XML parameters to command line arguments. Instead, ConfD just passes the raw XML on stdin to the program.

This statement is not allowed in 'tailf:action'.

tailf:interruptible Specifies whether the client can abort the execution of the executable.

tailf:interrupt This statement specifies which signal is sent to executable by ConfD in case the client terminates or aborts the execution.

If not specified, 'sigkill' is sent.

tailf:export *agent*

Makes this data model visible in the northbound interface 'agent'.

This statement makes it possible to have a data model visible through some northbound interface but not others. For example, if a MIB is used to generate a YANG module, the resulting YANG module can be exposed through SNMP only.

Use the special agent 'none' to make the data model completely hidden to all notherbound interfaces.

The agent can also be a free-form string. In this case, the data model will be visible to maapi applications using this string as its 'context'.

The *export* statement can be used in: *module*.

tailf:hidden *tag*

This statement can be used to hide a node from some, or all, northbound interfaces. All nodes with the same value are considered a hide group and are treated the same with regards to being visible or not in a northbound interface.

A node with an hidden property is not shown in the northbound user interfaces (CLI and Web UI) unless an 'unhide' operation has been performed in the user interface.

The hidden value 'full' indicates that the node should be hidden from all northbound interfaces, including programmatical interfaces such as NETCONF.

The value '*' is not valid.

A hide group can be unhidden only if this has been explicitly allowed in the confd.conf(5) daemon configuration.

Multiple hide groups can be specified by giving this statement multiple times. The node is shown if any of the specified hide groups has been given in the 'unhide' operation.

Note that if a mandatory node is hidden, a hook callback function (or similar) might be needed in order to set the element.

The *hidden* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:action*, *refine*, *tailf:symlink*, and *rpc*.

tailf:id *name*

This statement is used when old confspec models are translated to YANG. It needs to be present if systems deployed with data based on confspecs are updated to YANG based data models.

In confspec, the 'id' of a data model was a string that never would change, even if the namespace URI would change. It is not needed in YANG, since the namespace URi cannot change as a module is updated.

This statement is typically present in YANG modules generated by cs2yang. If no live upgrade needs to be done from a confspec based system to a YANG based system, this statement can be removed from such a generated module.

The *id* statement can be used in: *module*.

tailf:id-value *value*

This statement lets you specify a hard wired numerical id value to associate with the parent node. This id value is normally auto generated by confdc and is used when working with the ConfD API to refer to a tag name, to avoid expensive string comparison. Under certain rare circumstances this auto generated hash value may collide with a hash value generated for a node in another data model. Whenever such a collision occurs the ConfD daemon fails to start and instructs the developer to use the 'id-value' statement to resolve the collision.

A thorough discussion on id-value can be found in the section Hash Values and the id-value Statement in the YANG chapter in the User Guide.

The *id-value* statement can be used in: *module*, *leaf*, *leaf-list*, *list*, *container*, *rpc*, *identity*, *notification*, *choice*, *case*, and *tailf:action*.

tailf:indexed-view

This element can only be used if the list has a single key of an integer type.

It is used to signal that lists instances uses an indexed view, i.e., making it possible to insert a new list entry at a certain position. If a list entry is inserted at a certain position, list entries following this position are automatically renumbered by the system, if needed, to make room for the new entry.

This statement is mainly provided for backwards compatibility with confspecs. New data models should consider using YANG's ordered-by user statement instead.

The *indexed-view* statement can be used in: *list*.

The following substatements can be used:

tailf:auto-compact If an indexed-view list is marked with this statement, it means that the server will automatically renumber entires after a delete operation so that the list entries are strictly monotonically increasing, starting from 1, with no holes. New list entries can either be insterted anywhere in the list, or created at the end; but it is an error to try to create a list entry with a key that would result in a hole in the sequence.

For example, if the list has entries 1,2,3 it is an error to create entry 5, but correct to create 4.

tailf:info *text*

Contains a textual description of the definition, suitable for being presented to the CLI and WebUI users.

The first sentence of this textual description is used in the CLI as a summary, and displayed to the user when a short explanation is presented.

The 'description' statement is related, but targeted to the module reader, rather than the CLI or WebUI user.

The info string may contain a ';;' keyword. It is used in type descriptions for leafs when the builtin type info needs to be customized. A 'normal' info string describing a type is assumed to contain a short textual description. When ';;' is present it works as a delimiter where the text before the keyword is assumed to contain a short description and the text after the keyword a long(er) description. In the context of completion in the CLI the text will be nicely presented in two columns where both descriptions are aligned when displayed.

The *info* statement can be used in: *typedef*, *leaf*, *leaf-list*, *list*, *container*, *rpc*, *identity*, *type*, *enum*, *bit*, *length*, *pattern*, *range*, *refine*, *tailf:action*, *tailf:symlink*, and *tailf:cli-exit-command*.

tailf:info-html *text*

This statement works exactly as 'tailf:info', with the exception that it can contain HTML markup. The WebUI will display the string with the HTML markup, but the CLI will remove all HTML markup before displaying the string to the user. In most cases, using this statement avoids using special descriptions in webspecs and clispecs.

If this statement is present, 'tailf:info' cannot be given at the same time.

The *info-html* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *rpc*, *identity*, *tailf:action*, *tailf:symlink*, and *refine*.

tailf:java-class-name *name*

Used to give another name than the default name to generated Java classes. This statemement is typically used to avoid name conflicts in the Java classes.

The *java-class-name* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:junos-val-as-xml-tag

Internal extension to handle non-YANG JUNOS data models. Use only for key enumeration leafs.

The *junos-val-as-xml-tag* statement can be used in: *leaf*.

tailf:junos-val-with-prev-xml-tag

Internal extension to handle non-YANG JUNOS data models. Use only for keys where previous key is marked with 'tailf:junos-val-as-xml-tag'.

The *junos-val-with-prev-xml-tag* statement can be used in: *leaf*.

tailf:key-default *value*

Must be used for key leafs only.

Specifies a value that the CLI and WebUI will use when a list entry is created, and this key leaf is not given a value.

If one key leaf has a key-default value, all key leafs that follow this key leaf must also have key-default values.

The *key-default* statement can be used in: *leaf*.

tailf:link *target*

This statement specifies that the data node should be implemented as a link to another data node, called the target data node. This means that whenever the node is modified, the system modifies the target data node instead, and whenever the data node is read, the system returns the value of target data node.

Note that if the data node is a leaf, the target node MUST also be a leaf, and if the data node is a leaf-list, the target node MUST also be a leaf-list.

Note that the type of the data node MUST be the same as the target data node. Currently the compiler cannot check this.

The argument is an XPath absolute location path. If the target lies within lists, all keys must be specified. A key either has a value, or is a reference to a key in the path of the source node, using the function current() as starting point for an XPath location path. For example:

/a/b[k1='paul'][k2=current()/../k]/c

The *link* statement can be used in: *leaf* and *leaf-list*.

The following substatements can be used:

tailf:inherit-set-hook This statement specifies that a 'tailf:set-hook' statement should survive through symlinks. If set to true a set hook gets called as soon as the value is set via a symlink but also during commit. The normal behaviour is to only call the set hook during commit time.

tailf:lower-case

Use for config false leafs and leaf-lists only.

This extension serves as a hint to the system that the leaf's type has the implict pattern '[^A-Z]*', i.e., all strings returned by the data provider are lower case (in the 7-bit ASCII range).

The CLI uses this hint when it is run in case-insensitive mode to optimize the lookup calls towards the data provider.

The *lower-case* statement can be used in: *leaf* and *leaf-list*.

tailf:ncs-device-type *type*

Internal extension to tell NCS what type of device the data model is used for.

The *ncs-device-type* statement can be used in: *container*, *list*, *leaf*, *leaf-list*, and *refine*.

tailf:no-dependency

This optional statements can be used to explicitly say that a 'must' expression or a validation function is evaluated at every commit. Use this with care, since the overall performance of the system is impacted if this statement is used.

The *no-dependency* statement can be used in: *must* and *tailf:validate*.

tailf:non-strict-leafref

This statement can be used in leafs and leaf-lists similar to 'leafref', but allows reference to non-existing leafs, and allows reference from config to non-config.

This statement takes no argument, but expects the core YANG statement 'path' as a substatement. The function 'deref' cannot be used in the path, since it works on nodes of type leafref only.

The type of the leaf or leaf-list must be exactly the same as the type of the target.

This statement can be viewed as a substitute for a standard 'require-instance false' on leafrefs, which isn't allowed.

The CLI uses this statement to provide completion with existing values, and the WebUI uses it to provide a drop-down box with existing values.

The *non-strict-leafref* statement can be used in: *leaf* and *leaf-list*.

tailf:path-filters *value*

Used for type 'instance-identifier' only.

The argument is a space separated list of absolute or relative XPath expressions.

This statement declares that the instance-identifier value must match one of the specified paths, according to the following rules:

1. each XPath expression is evaluated, and returns a node set.

2. if there is no 'tailf:no-subtree-match' statement, the instance-identifier matches if it refers to a node in this node set, or if it refers to any descendant node of this node set.

3. if there is a 'tailf:no-subtree-match' statement, the instance-identifier matches if it refers to a node in this node set.

For example:

The value /a/b[key='k1']/c matches the XPath expression /a/b[key='k1']/c.

The value /a/b[key='k1']/c matches the XPath expression /a/b/c.

The value /a/b[key='k1']/c matches the XPath expression /a/b, if there is no 'tailf:no-subtree-match' statement.

The value /a/b[key='k1'] matches the XPath expression /a/b, if there is a 'tailf:no-subtree-match' statement.

The *path-filters* statement can be used in: *type*.

The following substatements can be used:

tailf:no-subtree-match See tailf:path-filters.

tailf:secondary-index *name*

This statement creates a secondary index with a given name in the parent list. The secondary index can be used to control the displayed sort order of the instances of the list.

Read more about sort order in 'The ConfD Command-Line Interface (CLI)' chapters in the User Guide, confd_lib_dp(3), and confd_lib_maapi(3).

NOTE: Currently secondary-index is not supported for config false data stored in CDB.

The *secondary-index* statement can be used in: *list*.

The following substatements can be used:

tailf:index-leafs This statement contains a space separated list of leaf names. Each such leaf must be a direct child to the list. The secondary index is kept sorted according to the values of these leafs.

tailf:sort-order

tailf:display-default-order Specifies that the list should be displayed sorted according to this secondary index in the show command.

If the list has more than one secondary index, 'display-default-order' must be present in one index only.

Used in J-, I- and C-style CLIs and WebUI.

tailf:snmp-delete-value *value*

This statement is used to define a value to be used in SNMP to delete an optional leaf. The argument to this statement is the special value. This special value must not be part of the value space for the YANG leaf.

If the optional leaf does not exists, reading it over SNMP returns 'noSuchInstance', unless the statement 'tailf:snmp-send-delete-value' is used, in which case the same value as used to delete the node is returned.

For example, the YANG leaf:

```
    leaf opt-int {
      type int32 {
        range '1..255';
      }
      tailf:snmp-delete-value 0 {
        tailf:snmp-send-delete-value;
      }
    }
```

can be mapped to a SMI object with syntax:

SYNTAX Integer32 (0..255)

Setting such an object to '0' over SNMP will delete the node from the datastore. If the node does not exsist, reading it over SNMP will return '0'.

The *snmp-delete-value* statement can be used in: *leaf*.

The following substatements can be used:

tailf:snmp-send-delete-value See tailf:snmp-delete-value.

tailf:snmp-exclude-object

Used when an SNMP MIB is generated from a YANG module, using the --generate-oids option to confdc.

If this statement is present, confdc will exclude this object from the resulting MIB.

The *snmp-exclude-object* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, and *refine*.

tailf:snmp-lax-type-check *value*

Normally, the ConfD MIB compiler checks that the data type of an SNMP object matches the data type of the corresponding YANG leaf. If both objects are writeble, the data types need to precisely match, but if the SNMP object is read-only, or if snmp-lax-type-check is set to 'true', the compiler accepts the object if the SNMP type's value space is a superset of the YANG type's value space.

If snmp-lax-type-check is true and the MIB object is writable, the SNMP agent will reject values outside the YANG data type range in runtime.

The *snmp-lax-type-check* statement can be used in: *leaf*.

tailf:snmp-mib-module-name *name*

Used when the YANG module is mapped to an SNMP module.

Specifies the name of the SNMP MIB module where the SNMP objects are defined.

This property is inherited by all child nodes.

The *snmp-mib-module-name* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *module*, and *refine*.

tailf:snmp-name *name*

Used when the YANG module is mapped to an SNMP module.

When the parent node is mapped to an SNMP object, this statement specifies the name of the SNMP object.

If the parent node is mapped to multiple SNMP objects, this statement can be given multiple times. The first statement specifies the primary table.

In a list, the argument is interpreted as:

[MIB-MODULE-NAME:]TABLE-NAME

For a leaf representing a table column, it is interpreted as:

[[MIB-MODULE-NAME:]TABLE-NAME:]NAME

For a leaf representing a scalar variable, it is interpreted as:

[MIB-MODULE-NAME:]NAME

If a YANG list is mapped to multiple SNMP tables, each such SNMP table must be specified with a 'tailf:snmp-name' statement. If the table is defined in another MIB than the MIB specified in 'tailf:snmp-mib-module-name', the MIB name must be specified in this argument.

A leaf in a list that is mapped to multiple SNMP tables must specify the name of the table it is mapped to if it is different from the primary table.

In the following example, a single YANG list 'interface' is mapped to the MIB tables ifTable, ifXTable, and ipv4InterfaceTable:

```
list interface {
  key index;
  tailf:snmp-name 'ifTable'; // primary table
  tailf:snmp-name 'ifXTable';
  tailf:snmp-name 'IP-MIB:ipv4InterfaceTable';
```

```
  leaf index {
    type int32;
  }
  leaf description {
    type string;
    tailf:snmp-name 'ifDescr';  // mapped to primary table
  }
  leaf name {
```

```
    type string;
    tailf:snmp-name 'ifXTable:ifName';
  }
  leaf ipv4-enable {
    type boolean;
    tailf:snmp-name
      'IP-MIB:ipv4InterfaceTable:ipv4InterfaceEnableStatus';
  }
  ...
}
```

When emitting a mib from yang, enum labels are used as-is if they follow the SMI rules for labels (no '.' or '_' characters and beginning with a lowercase letter). Any label that doesn't satisfy the SMI rules will be converted as follows:

An initial uppercase character will be downcased.

If the initial character is not a letter it will be prepended with an 'a'.

Any '.' or '_' characters elsewhere in the label will be substituted with '-' characters.

In the resulting label, any multiple '-' character sequence will be replaced with a single '-' character.

If this automatic conversion is not suitable, snmp-name can be used to specify the label to use when emitting a MIB.

The *snmp-name* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, *enum*, and *refine*.

tailf:snmp-ned-accessible-column *leaf-name*

The name or subid number of an accessible column that is instantiated in all table entries in a table. The column does not have to be writable. The SNMP NED will use this column when it uses GET-NEXT to loop through the list entries, and when doing existence tests.

If this column is not given, the SNMP NED uses the following algorithm:

1. If there is a RowStatus column, it will be used. 2. If an INDEX leaf is accessible, it will be used. 3. Otherwise, use the first accessible column returned by the SNMP agent.

The *snmp-ned-accessible-column* statement can be used in: *list*.

tailf:snmp-ned-delete-before-create

This statement is used in a list to make the SNMP NED always send deletes before creates. Normally, creates are sent before deletes.

The *snmp-ned-delete-before-create* statement can be used in: *list*.

tailf:snmp-ned-modification-dependent

This statement is used on all columns in a table that require the usage of the column marked with tailf:snmp-ned-set-before-row-modification.

This statement can be used on any column in a table where one leaf is marked with tailf:snmp-ned-set-before-row-modification, or a table that AUGMENTS such a table, or a table with a foreign index in such a table.

The *snmp-ned-modification-dependent* statement can be used in: *leaf*.

tailf:snmp-ned-recreate-when-modified

This statement is used in a list to make the SNMP NED delete and recreate the row when a column in the row is modified.

The *snmp-ned-recreate-when-modified* statement can be used in: *list*.

tailf:snmp-ned-set-before-row-modification *value*

If this statement is present on a leaf, it tells the SNMP NED that if a column in the row is modified, and it is marked with 'tailf:snmp-ned-modification-dependent', then the column marked with 'tailf:snmp-ned-set-before-modification' needs to be set to <value> before the other column is modified. After all such columns have been modified, the column marked with 'tailf:snmp-ned-set-before-modification' is reset to its initial value.

The *snmp-ned-set-before-row-modification* statement can be used in: *leaf*.

tailf:snmp-oid *oid*

Used when the YANG module is mapped to an SNMP module.

If this statement is present as a direct child to 'module', it indicates the top level OID for the module.

When the parent node is mapped to an SNMP object, this statement specifies the OID of the SNMP object. It may be either a full OID or just a suffix (a period, followed by an integer). In the latter case, a full OID must be given for some ancestor element.

NOTE: when this statement is set in a list, it refers to the OID of the correspondig table, not the table entry.

The *snmp-oid* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *tailf:symlink*, *module*, and *refine*.

tailf:snmp-row-status-column *value*

Used when an SNMP module is generated from the YANG module.

When the parent list node is mapped to an SNMP table, this statement specifies the column number of the generated RowStatus column. If it is not specified, the generated RowStatus column will be the last in the table.

The *snmp-row-status-column* statement can be used in: *list* and *refine*.

tailf:sort-order *how*

This statement can be used for 'ordered-by system' lists and leaf-lists only. It indicates in which way the list entries are sorted.

The *sort-order* statement can be used in: *list*, *leaf-list*, and *tailf:secondary-index*.

tailf:sort-priority *value*

This extension takes an integer parameter specifying the order and can be placed on leafs, containers, lists and leaf-lists. When showing, or getting configuration, leaf values will be returned in order of increasing sort-priority.

The default sort-priority is 0.

The *sort-priority* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, and *refine*.

tailf:step *value*

Used to further restrict the range of integer and decimal types. The argument is a positive integer or decimal value greater than zero. The allowed values for the type is further restricted to only those values that matches the expression:

'low' + n * 'step'

where 'low' is the lowest allowed value in the range, n is a non-negative integer.

For example, the following type:

```
type int32 {
  range '-2 .. 9' {
    tailf:step 3;
  }
}
```

```
has the value space { -2, 1, 4, 7 }
```

The *step* statement can be used in: *range*.

tailf:structure *name*

Internal extension to define a data structure without any semantics attached.

The *structure* statement can be used in: *module* and *submodule*.

tailf:suppress-echo *value*

If this statetement is set to 'true', leafs of this type will not have their values echoed when input in the webui or when the CLI prompts for the value. The value will also not be included in the audit log in clear text but will appear as ***.

The *suppress-echo* statement can be used in: *typedef*, *leaf*, and *leaf-list*.

tailf:symlink *name*

DEPRECATED: Use tailf:link instead. There are no plans to remove tailf:symlink.

This statement defines a 'symbolic link' from a node to some other node. The argument is the name of the new node, and the mandatory substatement 'tailf:path' points to the node which is linked to.

The *symlink* statement can be used in: *list*, *container*, *module*, *submodule*, *augment*, and *case*.

The following substatements can be used:

tailf:alt-name

tailf:cli-add-mode

tailf:cli-allow-join-with-key

tailf:cli-allow-join-with-value

tailf:cli-allow-key-abbreviation

tailf:cli-allow-range

tailf:cli-allow-wildcard

tailf:cli-autowizard

tailf:cli-boolean-no

tailf:cli-break-sequence-commands

tailf:cli-column-align

tailf:cli-column-stats

tailf:cli-column-width

tailf:cli-compact-stats

tailf:cli-compact-syntax

tailf:cli-completion-actionpoint

tailf:cli-custom-error

tailf:cli-custom-range

tailf:cli-custom-range-actionpoint

tailf:cli-custom-range-enumerator

tailf:cli-delayed-auto-commit

tailf:cli-delete-container-on-delete

tailf:cli-delete-when-empty

tailf:cli-diff-dependency

tailf:cli-disabled-info

tailf:cli-disallow-value

tailf:cli-display-empty-config

tailf:cli-display-separated

tailf:cli-drop-node-name

tailf:cli-no-keyword

tailf:cli-enforce-table

tailf:cli-embed-no-on-delete

tailf:cli-exit-command

tailf:cli-explicit-exit

tailf:cli-expose-key-name

tailf:cli-expose-ns-prefix

tailf:cli-flat-list-syntax

tailf:cli-flatten-container

tailf:cli-full-command

tailf:cli-full-no

tailf:cli-full-show-path

tailf:cli-hide-in-submode

tailf:cli-ignore-modified

tailf:cli-incomplete-command

tailf:cli-incomplete-no

tailf:cli-incomplete-show-path

tailf:cli-instance-info-leafs

tailf:cli-key-format

tailf:cli-list-syntax

tailf:cli-min-column-width

tailf:cli-mode-name

tailf:cli-mode-name-actionpoint

tailf:cli-multi-value

tailf:cli-multi-word-key

tailf:cli-multi-line-prompt

tailf:cli-no-key-completion

tailf:cli-no-match-completion

tailf:cli-no-name-on-delete

tailf:cli-no-value-on-delete

tailf:cli-oper-info

tailf:cli-optional-in-sequence

tailf:cli-prefix-key

tailf:cli-preformatted

tailf:cli-range-delimiters

tailf:cli-range-list-syntax

tailf:cli-recursive-delete

tailf:cli-remove-before-change

tailf:cli-reset-container

tailf:cli-run-template

tailf:cli-run-template-enter

tailf:cli-run-template-footer

tailf:cli-run-template-legend

tailf:cli-sequence-commands

tailf:cli-show-config

tailf:cli-show-no

tailf:cli-show-order-tag

tailf:cli-show-order-taglist

tailf:cli-show-template

tailf:cli-show-template-enter

tailf:cli-show-template-footer

tailf:cli-show-template-legend

tailf:cli-show-with-default

tailf:cli-strict-leafref

tailf:cli-suppress-key-abbreviation

tailf:cli-suppress-key-sort

tailf:cli-suppress-list-no

tailf:cli-suppress-mode

tailf:cli-suppress-no

tailf:cli-suppress-range

tailf:cli-suppress-shortenabled

tailf:cli-suppress-show-conf-path

tailf:cli-suppress-show-match

tailf:cli-suppress-show-path

tailf:cli-suppress-silent-no

tailf:cli-suppress-validation-warning-prompt

tailf:cli-suppress-wildcard

tailf:cli-table-footer

tailf:cli-table-legend

tailf:cli-trim-default

tailf:cli-value-display-template

tailf:display-when

tailf:hidden

tailf:inherit-set-hook This statement specifies that a 'tailf:set-hook' statement should survive through symlinks. If set to true a set hook gets called as soon as the value is set via a symlink but also during commit. The normal behaviour is to only call the set hook during commit time.

tailf:info

tailf:info-html

tailf:path This statement specifies which node a symlink points to.

The textual format of a symlink is an XPath absolute location path. If the target lies within lists, all keys must be specified. A key either has a value, or is a reference to a key in the path of the source node, using the function current() as starting point for an XPath location path. For example:

/a/b[k1='paul'][k2=current()/../k]/c

tailf:snmp-exclude-object

tailf:snmp-name

tailf:snmp-oid

tailf:sort-priority

tailf:typepoint *id*

If a typedef, leaf, or leaf-list has a 'typepoint' statement, a user-defined type is specified, as opposed to a derivation or specification of an existing type. The implementation of a user-defined type must be provided in the form of a shared object with C callback functions that is loaded into the ConfD daemon at startup time. Read more about user-defined types in the confd_types(3) manual page.

The argument defines the ID associated with a typepoint. This ID is provided by the shared object, and used by the ConfD daemon to locate the implementation of a specific user-defined type.

The *typepoint* statement can be used in: *typedef*, *leaf*, and *leaf-list*.

tailf:unique-selector *context-path*

The standard YANG statement 'unique' can be used to check for uniqueness within a single list only. Specifically, it cannot be used to check for uniqueness of leafs within a sublist.

For example:

```
container a {
  list b {
    ...
    unique 'server/ip server/port';
    list server {
      ...
      leaf ip { ... };
      leaf port { ... };
    }
  }
}
```

The unique expression above is not legal. The intention is that there must not be any two 'server' entries in any 'b' with the same combination of ip and port. This would be illegal:

<a> <name>b1</name> <server> <ip>10.0.0.1</ip> <port>80</port> </server>   <name>b2</name> <server> <ip>10.0.0.1</ip> <port>80</port> </server>

With 'tailf:unique-selector' and 'tailf:unique-leaf', this kind of constraint can be defined.

The argument to 'tailf:unique-selector' is an XPath descendant location path (matches the rule 'descendant-schema-nodeid' in RFC 6020). The first node in the path MUST be a list node, and it MUST be defined in the same module as the tailf:unique-selector. For example, the following is illegal:

```
module y {
  ...
  import x {
    prefix x;
  }
  tailf:unique-selector '/x:server' { // illegal
    ...
  }
}
```

For each instance of the node where the selector is defined, it is evaluated, and for each node selected by the selector, a tuple is constructed by evaluating the 'tailf:unique-leaf' expression. All such tuples must be unique. If a 'tailf:unique-leaf' expression refers to a non-existing leaf, the corresponding tuple is ignored.

In the example above, the unique expression can be replaced by:

```
container a {
  tailf:unique-selector 'b/server' {
    tailf:unique-leaf 'ip';
    tailf:unique-leaf 'port';
  }
  list b {
    ...
  }
}
```

For each container 'a', the XPath expression 'b/server' is evaluated. For each such server, a 2-tuple is constructed with the 'ip' and 'port' leafs. Each such 2-tuple is guaranteed to be unique.

The *unique-selector* statement can be used in: *module*, *submodule*, *grouping*, *augment*, *container*, and *list*.

The following substatements can be used:

tailf:unique-leaf See 'tailf:unique-selector' for a description of how this statement is used.

The argument is an XPath descendant location path (matches the rule 'descendant-schema-nodeid' in RFC 6020), and it MUST refer to a leaf.

tailf:validate *id*

Identifies a validation callback which is invoked when a configuration value is to be validated. The callback validates a value and typically checks it towards other values in the data store. Validation callbacks are used when the YANG built-in validation constructs ('must', 'unique') are not expressive enough.

Callbacks use the API described in confd_lib_maapi(3) to access whatever other configuration values needed to perform the validation.

Validation callbacks are typically assigned to individual nodes in the data model, but it may be feasible to use a single validation callback on a root node. In that case the callback is responsible for validation of all values and their relationships throughout the data store.

The 'validate' statment should in almost all cases have a 'tailf:dependency' substatement. If such a statement is not given, the validate function is evaluated at every commit, leading to overall performance degradation.

If the 'validate' statement is defined in a 'must' statement, validation callback is called instead of evaluating the must expression. This is useful if the evaluation of the must statement uses too much resources, and the condition expressed with the must statement is easier to check with a validation callback function.

The *validate* statement can be used in: *leaf*, *leaf-list*, *list*, *container*, *grouping*, *refine*, and *must*.

The following substatements can be used:

tailf:call-once This optional statement can be used only if the parent statement is a list. If 'call-once' is 'true'. the validation callback is only called once even though there exists many list entries in the data store. This is useful if we have a huge amount of instances or if values assigned to each instance have to be validated in comparison with its siblings.

tailf:dependency

tailf:opaque Defines an opaque string which is passed to the callback function in the context.

tailf:internal For internal ConfD / NCS use only.

tailf:priority This extension takes an integer parameter specifying the order validation code will be evaluated, in order of increasing priority.

The default priority is 0.

tailf:value-length *value*

Used only for the types: yang:object-identifier yang:object-identifier-128 yang:phys-address yang:hex-string tailf:hex-list tailf:octet-list xs:hexBinary

This type restriction is used to limit the length of the value-space value of the type. Note that since all these types are derived from 'string', the standard 'length' statement restricts the lexical representation of the value.

The argument is a length expression string, with the same syntax as for the standard YANG 'length' statement.

The *value-length* statement can be used in: *type*.

tailf:writable *value*

This extension makes operational data (i.e., config false data) writable. Only valid for leafs.

The *writable* statement can be used in: *leaf*.

tailf:xpath-root *value*

Internal extension to 'chroot' XPath expressions

The *xpath-root* statement can be used in: *must*, *when*, *path*, *tailf:display-when*, and *tailf:cli-diff-dependency*.

YANG TYPES

aes-cfb-128-encrypted-string

The aes-cfb-128-encrypted-string works exactly like des3-cbc-encrypted-string but AES/128bits in CFB mode is used to encrypt the string. The prefix for encrypted values is '4'.

des3-cbc-encrypted-string

The des3-cbc-encrypted-string type automatically encrypts a value adhering to this type using DES in CBC mode followed by a base64 conversion. If the value isn't encrypted already, that is.

This is best explained using an example. Suppose we have a leaf:

```
  leaf enc {
      type tailf:des3-cbc-encrypted-string;
  }
```

A valid configuration is:

<enc>0In god we trust.</enc>

The '0' prefix signals that this is plain text. When a plain text value is received by the server, the value is DES3/Base64 encrypted, and the string '3' is prepended. The resulting string is stored in the configuration data store.

When a value of this type is read, the encrypted value is always returned. In the example above, the following value could be returned:

<enc>3lyPjszaQq4EVqK7OPOxybQ==</enc>

If a value starting with '3' is received, the server knows that the value is already encrypted, and stores it as is in the data store.

A value adhering to this type must have a '0' or a '3' prefix.

ConfD uses a configurable set of encryption keys to encrypt the string. For details, see 'encryptedStrings' in the confd.conf(5) manual page.

hex-list

DEPRECATED: Use yang:hex-string instead. There are no plans to remove tailf:hex-list.

A list of colon-separated hexa-decimal octets e.g. '4F:4C:41:71'.

The statement tailf:value-length can be used to restrict the number of octets. Note that using the 'length' restriction limits the number of characters in the lexical representation.

ip-address-and-prefix-length

The ip-address-and-prefix-length type represents a combination of an IP address and a prefix length and is IP version neutral. The format of the textual representations implies the IP version.

ipv4-address-and-prefix-length

The ipv4-address-and-prefix-length type represents a combination of an IPv4 address and a prefix length. The prefix length is given by the number following the slash character and must be less than or equal to 32.

ipv6-address-and-prefix-length

The ipv6-address-and-prefix-length type represents a combination of an IPv6 address and a prefix length. The prefix length is given by the number following the slash character and must be less than or equal to 128.

md5-digest-string

The md5-digest-string type automatically computes a MD5 digest for a value adhering to this type.

This is best explained using an example. Suppose we have a leaf:

```
  leaf key {
      type tailf:md5-digest-string;
  }
```

A valid configuration is:

<key>0In god we trust.</key>

The '0' prefix signals that this is plain text. When a plain text value is received by the server, an MD5 digest is calculated, and the string '1<salt>$' is prepended to the result, where <salt> is a random eight character salt used to generate the digest. This value is stored in the configuration data store.

When a value of this type is read, the computed MD5 value is always returned. In the example above, the following value could be returned:

<key>1fB$ndk2z/PIS0S1SvzWLqTJb.</key>

If a value starting with '1' is received, the server knows that the value already represents an MD5 digest, and stores it as is in the data store.

A value adhering to this type must have a '0' or a '1<salt>$' prefix.

If a default value is specified, it must have a '1<salt>$' prefix.

The digest algorithm used is the same as the md5 crypt function used for encrypting passwords for various UNIX systems, see e.g. http://www.freebsd.org/cgi/cvsweb.cgi/~checkout/~/src/lib/libcrypt/crypt.c

octet-list

A list of dot-separated octets e.g. '192.168.255.1.0'.

The statement tailf:value-length can be used to restrict the number of octets. Note that using the 'length' restriction limits the number of characters in the lexical representation.

sha-256-digest-string

The sha-256-digest-string type automatically computes a SHA-256 digest for a value adhering to this type.

A value of this type matches one of the forms:

0<clear text password> 5<salt>$<password hash> 5rounds=<number>$<salt>$<password hash>

The '0' prefix signals that this is plain text. When a plain text value is received by the server, a SHA-256 digest is calculated, and the string '5<salt>$' is prepended to the result, where <salt> is a random 16 character salt used to generate the digest. This value is stored in the configuration data store. The algorithm can be tuned via the /confdConfig/cryptHash/rounds parameter, which if set to a number other than the default will cause '5rounds=<number>$<salt>$' to be prepended instead of only '5<salt>$'.

If a value starting with '5' is received, the server knows that the value already represents a SHA-256 digest, and stores it as is in the data store.

If a default value is specified, it must have a '5' prefix.

The digest algorithm used is the same as the SHA-256 crypt function used for encrypting passwords for various UNIX systems, see e.g. http://www.akkadia.org/drepper/SHA-crypt.txt

sha-512-digest-string

The sha-512-digest-string type automatically computes a SHA-512 digest for a value adhering to this type.

A value of this type matches one of the forms:

0<clear text password> 6<salt>$<password hash> 6rounds=<number>$<salt>$<password hash>

The '0' prefix signals that this is plain text. When a plain text value is received by the server, a SHA-512 digest is calculated, and the string '6<salt>$' is prepended to the result, where <salt> is a random 16 character salt used to generate the digest. This value is stored in the configuration data store. The algorithm can be tuned via the /confdConfig/cryptHash/rounds parameter, which if set to a number other than the default will cause '6rounds=<number>$<salt>$' to be prepended instead of only '6<salt>$'.

If a value starting with '6' is received, the server knows that the value already represents a SHA-512 digest, and stores it as is in the data store.

If a default value is specified, it must have a '6' prefix.

The digest algorithm used is the same as the SHA-512 crypt function used for encrypting passwords for various UNIX systems, see e.g. http://www.akkadia.org/drepper/SHA-crypt.txt

size

A value that represents a number of bytes. An example could be S1G8M7K956B; meaning 1GB + 8MB + 7KB + 956B = 1082138556 bytes. The value must start with an S. Any byte magnifier can be left out, e.g. S1K1B equals 1025 bytes. The order is significant though, i.e. S1B56G is not a valid byte size.

In ConfD, a 'size' value is represented as an uint64.

XPATH FUNCTIONS

This section describes XPath functions that can be used for example in "must" expressions in YANG modules.

node-set **deref**(*node-set*)

The `deref()` function follows the reference defined by the first node in document order in the argument node-set, and returns the nodes it refers to.

If the first argument node is an instance-identifier, the function returns a node-set that contains the single node that the instance identifier refers to, if it exists. If no such node exists, an empty node-set is returned.

If the first argument node is a leafref, the function returns a node-set that contains the nodes that the leafref refers to.

If the first argument node is of any other type, an empty node-set is returned.

bool **re-match**(*string*, *string*)

The `re-match()` function returns `true` if the string in the first argument matches the regular expression in the second argument; otherwise it returns `false`.

For example: `re-match('1.22.333', '\d{1,3}\.\d{1,3}\.\d{1,3}')` returns `true`. To count all logical interfaces called eth0.*number*: `count(/sys/ifc[re-match(name,'eth0\.\d+')])`.

The regular expressions used are the XML Schema regular expressions, as specified by W3C in http://www.w3.org/TR/xmlschema-2/#regexs. Note that this includes implicit anchoring of the regular expression at the head and tail, i.e. if you want to match an interface that has a name that starts with 'eth' then the regular expression must be `'eth.*'`.

number **string-compare**(*string*, *string*)

The `string-compare()` function returns -1, 0, or 1 depending on whether the value of the string of the first argument is respectively less than, equal to, or greater than the value of the string of the second argument.

number **compare**(*Expression*, *Expression*)

The `compare()` function returns -1, 0, or 1 depending on whether the value of the first argument is respectively less than, equal to, or greater than the value of the second argument.

The expressions are evaluated in a special way: If they both are XPath constants they are compared using the `string-compare()` function. But, more interestingly, if the expressions results in node-sets with at least one node, and that node is an existing leaf that leafs value is compared with the other expression, and if the other expression is a constant that expression is converted to an internal value with the same type as the expression that resulted in a leaf. Thus making it possible to order values based on the internal

representation rather than the string representation. For example, given a leaf:

```
leaf foo {
  type enumeration {
    enum ccc;
    enum bbb;
    enum aaa;
  }
}
```

it would be possible to call `compare(foo, 'bbb')` (which, for example, would return -1 if foo='ccc'). Or to have a must expression like this: `must "compare(.,'bbb') >= 0";` which would require foo to be set to 'bbb' or 'aaa'.

If one of the expressions result in an empty node-set, a non-leaf node, or if the constant can't be converted to the other expressions type then `NaN` is returned.

number **min**(*node-set*)

Returns the numerically smallest number in the node-set, or `NaN` if the node-set is empty.

number **max**(*node-set*)

Returns the numerically largest number in the node-set, or `NaN` if the node-set is empty.

number **avg**(*node-set*)

Returns the numerical average of the node-set, or `NaN` if the node-set is empty, or if any numerical conversion of a node failed.

number **band**(*number*, *number*)

Returns the result of bitwise AND:ing the two numbers. Unless the numbers are integers NaN will be returned.

number **bor**(*number*, *number*)

Returns the result of bitwise OR:ing the two numbers. Unless the numbers are integers NaN will be returned.

number **bxor**(*number*, *number*)

Returns the result of bitwise Exclusive OR:ing the two numbers. Unless the numbers are integers NaN will be returned.

number **bnot**(*number*)

Returns the result of bitwise NOT on number. Unless the number is an integer NaN will be returned.

node-set **sort-by**(*node-set*, *string*)

The `sort-by()` function makes it possible to order a node-set according to a secondary index (see the tailf:secondary-index extension). The first argument must be an expression that evaluates to a node-set, where the nodes in the node-set are all list instances of the same list. The second argument must be the name of an existing secondary index on that list. For example given the YANG model:

```
container sys {
  list host {
    key name;
    unique number;
    tailf:secondary-index number {
      tailf:index-leafs "number";
    }
    leaf name {
      type string;
```

```
    }
    leaf number {
      type uint32;
      mandatory true;
    }
    leaf enabled {
      type boolean;
      default true;
    }
  ...
  }
}
```

The expression `sort-by(/sys/host,"number")` would result in all hosts, sorted by their number. And the expression, `sort-by(/sys/host[enabled='true'],"number")` would result in all enabled hosts, sorted by number. Note also that since the function returns a node-set it is also legal to add location steps to the result. I.e. the expression `sort-by(/sys/host[enabled='true'],"number")/name` results in all host names sorted by the hosts number.

SEE ALSO

tailf_yang_cli_extensions(5)	Tail-f YANG CLI extensions
The ConfD User Guide	
confdc(1)	Confdc compiler

Glossary

Many of the following entries refer to this example data model, which models a list of "servers":

Example 148. The servers YANG model

```
module servers {
    namespace "http://www.example.com/ns/servers";
    prefix srv;

    import ietf-inet-types {
        prefix inet;
    }

    import dataTypes {
        prefix dt;
    }

    container servers {
        list server {
            key "ip port";
            max-elements 64;
            leaf ip {
                type inet:ip-address;
            }
            leaf port {
                type inet:port-number;
            }
            leaf counters {
                type dt:countersType;
                mandatory true;
                config false;
            }
        }
    }
}
```

AAA

AAA stands for *authentication*, *authorization* and *accounting*.

ConfD requires the data model defined by the namespace `http://tail-f.com/ns/aaa/VERSION` to be loaded. Currently valid values for *VERSION* are 1.0 and 1.1. The data found there is used to authenticate users and authorize access for users. We usually refer to this namespace as the AAA namespace.

Accounting

Accounting refers to the tracking of the consumption of network resources by users. This information may be used for management, planning, billing, or other purposes. Typical information that is gathered in accounting is the identity of the user, the nature of the service delivered, when the service began, and when it ended.
See Also AAA.

Agent

See Management Agent.

Annotation

When compiling a YANG module, it can be annotated with callpoints, actionpoints and validation elements from a separate annotation file. This is useful for example when implementing a standard YANG module, without modifying the original file.

Authentication

Authentication refers to the confirmation that a user who is requesting services is a valid user of the network services requested. Authentication is accomplished via the presentation of an identity and credentials. Examples of types of credentials are passwords and digital certificates.
See Also AAA.

Authorization

Authorization refers to the granting of specific types of service (including "no service") to a user, based on authentication, what services they are requesting, and the current system state. Authorization may be based on restrictions, for example time-of-day restrictions, or physical location restrictions, or restrictions against multiple logins by the same user. Authorization determines the nature of the service which is granted to a user.

In ConfD all actions are authorized by reading the authorization data found in the AAA namespace `http://tail-f.com/ns/aaa/version`.
See Also AAA.

Backplane

See Management Backplane.

Candidate datastore

The candidate datastore (or just candidate) is one of the three configuration datastores in ConfD. It holds changes to the configuration before they are committed to the *running datastore*.

CDB

CDB is the built-in configuration database provided by ConfD. It is possible to use an *external database* to store the configuration, or to use CDB, as well as a combination of both.

CDB C API

This is the C API towards CDB. It contains functions to read configuration data, read and write operational data, and subscribe to changes in configuration data.

CDB session

A program which uses the CDB C API needs to establish CDB sessions to read configuration data, or read/write operational data. These are short-lived sessions that are established through a call to `cdb_start_session()`.

The entire configuration part of CDB is locked for writing while any CDB read session is active.

CDB upgrade

An upgrade is the operation of adapting an existing configuration to a newer version of the configuration schemas.

Confspec

Confspec is a Tail-f proprietary data modelling language. YANG is a better standards based technology to achieve the same thing. Confspecs are no longer supported.

Configuration

A configuration is an instantiated data model. The data model defines the layout of the configuration. An example of a configuration which adheres to the data model in Example 148, “The servers YANG model” is:

```
<servers>
  <server>
    <name>www</name>
    <ip>192.168.128.1</ip>
    <port>80</port>
  </server>
  <server>
    <name>pop</name>
    <ip>0.0.0.0</ip>
```

```
    <port>110</port>
  </server>
</servers>
```

The data model also defines "non-configuration" data, also known as statistical or operational data. This data is not part of the configuration.

Daemon

A daemon is a UNIX process that runs in the background. In the ConfD documentation we refer to the "ConfD daemon" meaning the "confd" process. We also refer to external database processes as daemons. Typically in a deployment scenario, the programs implementing the external database callbacks will run as daemons.

Data provider

Southbound of ConfD we may have several data providers. These can either be *daemons* implementing the ConfD "data callbacks" or the ConfD database *CDB*.

A data provider has the following responsibilities:

- Read and return data when ConfD requests data to be read
- Write data into its store when ConfD requests it to write data
- Obey the two-phase commit protocol used by ConfD when writing data

Data store

See Data provider.

Export

When a data model is compiled, it can be exported to selected northbound interfaces, instead of being visible to all.

External database

We may choose to store all configuration data in an external database, or in plain configuration files. From ConfD's point of view, we then have the configuration in an external database and no data is kept in CDB. There can be multiple external databases simultaneously connected to ConfD. It is also possible to use a combination of CDB and external database(s).

The C or Java code which implements an "external database" must adhere strictly to the ConfD transaction protocol. This is done by implementing a specific set of callback C functions. This could be viewed as the equivalent of implementing instrumentation functions in an SNMP agent.

Initialization file

An XML file (with suffix .xml) used for initializing the part of the *configuration* which is stored in the *CDB* database.

Instrumentation

This is the task of implementing data callback functions for an *external database*.

Keypath

A keypath is a string that uniquely identifies a node in a Configuration. A keypath is an (older) alternative syntax for a YANG instance-identifier.

The following are examples of keypaths

`/servers/server{www}/port`	which uniquely identifies the data element: `<port>80</port>` in the first server in Configuration.
`/servers`	which identifies the top level `servers` container element.

A keypath is a path down the XML tree. Which path to choose down the tree, i.e. which list entry to choose, must be indicated with a `{key}` notation.

The keypath `/servers/server{www}/port` is equivalent to the instance-identifier `/servers/server[name="www"]/port`.

We also have a different notation which is used in the CDB C API to identify data elements. The CDB C API contains a function `cdb_num_instances()` which returns an integer. In Configuration we had two different servers, thus `cdb_num_instances()` would return 2 and the keypath `/servers/server[1]/port` would uniquely identify the `port` element in the second `server` element. A keypath that uses the *[index]* notation in the path is only valid for the current CDB session.

In the C callback APIs, keypaths in the form of hashed keypaths are represented by the C type confd_hkeypath_t.

Management Agent

A Management Agent, or simply Agent, is a software entity which terminates some management protocol and provides a view of the managed system. The Agents in ConfD are NETCONF, SNMP, CLI, and Web UI, and proprietary agents that utilize the *MAAPI* API.

Management Backplane

This is the layer of software inside ConfD which sits between the northbound agent interfaces and the southbound data providers. The main task of the management backplane is to multiplex and format data between the northbound and the southbound interfaces through transactions.

MAAPI

MAAPI - The Management Agent API - is an API which is used to connect to the ConfD transaction system. ConfD provides MAAPI bindings for C and Java. MAAPI is a northbound interface API which can be used to perform all read and write operations towards ConfD. It is possible to implement any proprietary configuration Agent with MAAPI.

MAAPI is also used to implement CLI wizards and semantic validation of configuration data in C.

Path filter

Path filters restrict the set of valid elements to a subtree. The value of an element with the type confd:objectRef is a pointer to another element, and the set of elements to which the value is allowed to point can be restricted by specifying a list of path filters, in XPath syntax.

Likewise, the CLI **show** command can be restricted by a path filter (as a space-separated list of path items).

Running datastore

The running datastore is one of the three configuration datastores in ConfD. It contains the currently active *configuration*.

Schema

A schema defines the structure of data. A schema in ConfD is represented by YANG modules, compiled to a schema file having the file name suffix ".fxs".

Session

See CDB session or User session.

Startup datastore

The startup datastore is one of the three configuration datastores in ConfD. It contains a *configuration* to be read by the device each time it reboots.

Subscription

Programs communicating with ConfD using one of the APIs can ask to be notified of certain events, by setting up a subscription. Subscription functionality is available in the CDB C API for notification of CDB configuration changes and for the asynchronous events described in the "Notifications" chapter.

Tagpath

A tagpath is a string and it is similar to a *keypath* with the exception that there are no keys in a tagpath. For example the string */servers/server/mask* is a tagpath whereas the string */servers/server{www}/mask* is a keypath.

The keypath above uniquely identifies a single instance of a particular mask in a particular server, namely the server named "www", whereas the tagpath above identifies all masks in all servers in the *configuration*.

Target device

By target device, or just "device", we mean the entire embedded box that is being built, such as the physical router if it is a router that is being configured by ConfD.

Transaction

ConfD implements all writes towards all three possible configuration stores, *start-up*, *running* and the *candidate* as two-phase commit transaction.

A configuration may be stored in several databases. We can have some part of the configuration in CDB and other parts in several *external databases*. A commit operation may span over several databases and we always want to ensure that all participants, i.e., all involved databases are ready to commit. Thus the C API to implement an external database always requires the external database to implement several callback C functions which will be called during the different phases of the transaction.

A transaction is always associated with a corresponding *user session*.

Transformations

A transformation is used when we have a data model which we do not want to expose through the northbound Agent interfaces. We hide the undesired data model, and expose another model which then must *transform* data to and from the hidden model.

User session

A user session corresponds directly to an SSH/SSL session from a management station to ConfD. A user session is associated with such data as the IP address of the management station and the user name of the user who logged in to ConfD, whether through NETCONF, the CLI or the Web UI.

Validation

Validation is the process of ensuring the correctness of the input configuration data. Syntactic validation is the first phase and it ensures type correctness and all checks that are possible to express in the YANG model. Semantic validation is the second phase and it requires application specific knowledge. The programmer has to write C or Java code which performs the semantic validation.

The command **confdc -x** can be used for syntactic validation of *initialization files*.

XPath

XPath is a language for selecting parts of an XML document. ConfD uses XPath (version 1.0) in its full form in NETCONF requests, as well as a very restricted subset to specify *path filters* in elements of the confd:objectRef type.