Clixon

Release 7.0

Olof Hagsand

Apr 03, 2024

TABLE OF CONTENTS

Clixon is a YANG-based configuration manager, with interactive CLI, NETCONF and RESTCONF interfaces, an embedded database and a transaction mechanism.

CHAPTER

ONE

1 OVERVIEW

Clixon is a configuration management framework used by networking devices and other computer systems. Clixon provides a datastore, CLI, NETCONF and RESTCONF interfaces all defined by YANG.

Clixon links:

- Source code at github.
- Project.
- Docs.

Most of the projects using Clixon are for networking devices. But Clixon can be used for other YANG-based systems as well due to a modular and pluggable architecture.

Clixon has a transaction mechanism that ensures configuration operations are atomic. It also features a generated interactive command-line interface using CLIgen.

The goal of Clixon is to provide a useful, production-grade, scalable and free YANG based configuration tool.

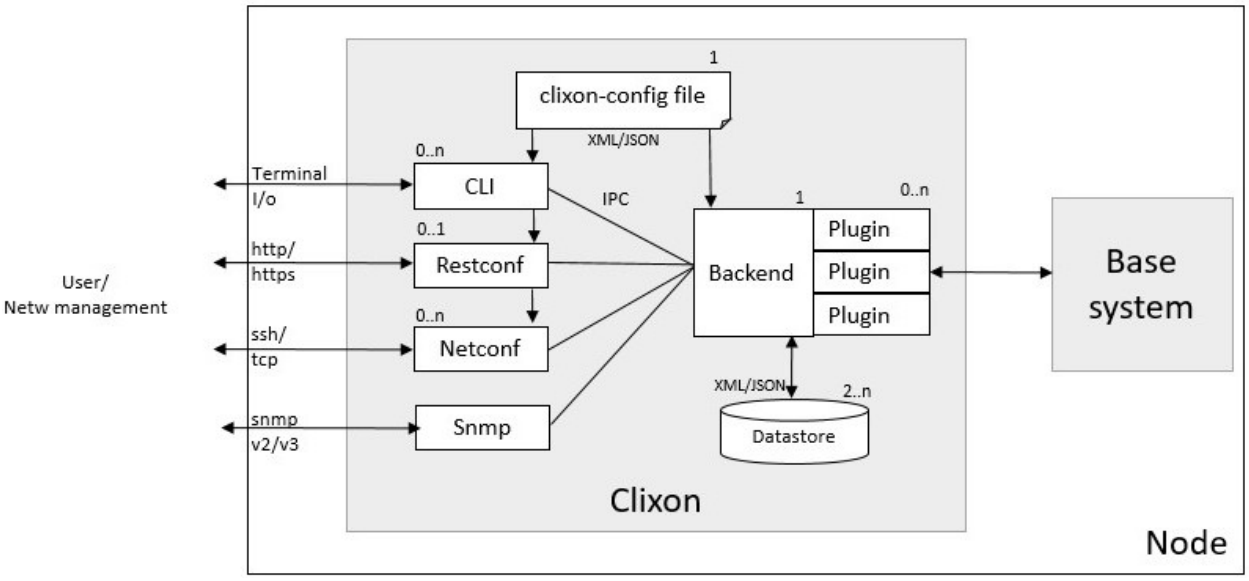
Clixon is open-source and dual licensed. Either Apache License, Version 2.0 or GNU General Public License Version 2.

1.1 1.1 System Architecture

Clixon provides YANG functionality with Netconf, Restconf and CLI that can be integrated with an existing "base system" in several ways. The integrations are:

- A *plugin* integration where clixon handles all user interaction with the base system using backend plugins. This is the _primary_ Clixon usage model.
- A *client* integration where the base system uses clixon for configurations as a "side-car". There is some ongoing work to make Clixon also work for this usage.

1.1.1 1.1.1 Plugin integration



This describes how to integrate a base system with Clixon using plugins.

The Clixon architecture consists of a backend daemon with configuration datastores and a set of internal clients: cli, restconf, netconf and snmp.

The clients provide frontend interfaces to users of the system, such as a Network Management System (NMS) or an interactive human user. The external interfaces include interactive CLI, RESTCONF over HTTP/HTTPS, and XML NETCONF over TCP or SSH. Internally, the clients and backend communicate over an inter-process communication (IPC) bus via NETCONF over a UNIX socket. It is possible to run over an INET socket as well.

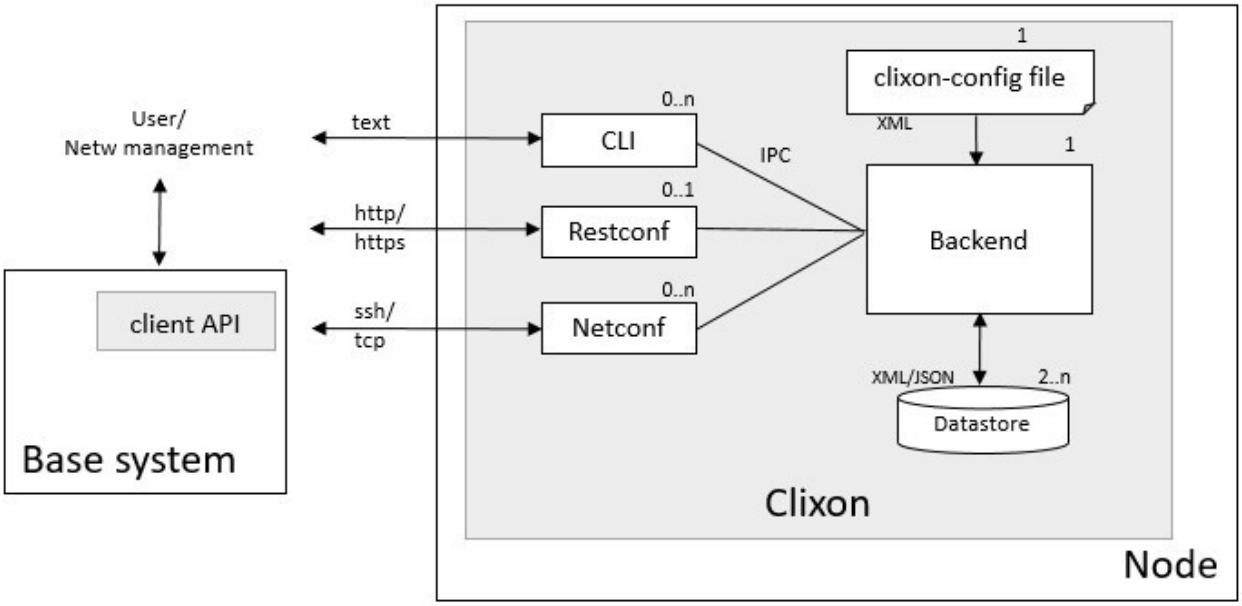
The backend manages configuration datastores and implements a transaction mechanism for configuration operations (eg, create, read, update, delete) . The datastore supports candidate, running and startup configurations.

A system integrating Clixon using plugins, typically starts with a set of YANG specifications. Backend plugins are written that interact with the base system. The plugins are written in C using the Clixon API and a set of plugin callbacks. The main callback is a transaction callback, where you specify how configuration changes are made to your system.

You can also design an interactive CLI using CLIgen, where you specify the CLI commands and write CLI plugins. You will have to write CLI rules, but Clixon can generate the configuration part of the CLI, including set, delete, show commands for a specific syntax.

Notifications (streams) are supported both for CLI, NETCONF and RESTCONF clients.

1.1.2 1.1.2 Client integration



In a client architecture, the base system keeps existing APIs and only YANG-based configurations are exposed via Clixon. The base system acts as a clixon client and uses the clixon client module to subscribe to configuration events using Netconf message passing.

In comparison, the tighter plugin architecture uses dynamically loaded plugins, callbacks and a shared datastore. See *clixon client api* for more details.

1.2 1.2 Platforms

Clixon supports GNU/Linux, FreeBSD and Docker. MacOS may work. Linux platforms include Ubuntu, Alpine, Centos, and Raspian. CPU architectures include x86_64, i686, and ARM32.

1.3 1.3 Standards

Clixon supports standards including YANG, NETCONF, RESTCONF, XML and XPath. See *Standards section* for more details.

1.4 1.4 How to get Clixon

Get the Clixon source code from Github:

```
git clone https://github.com/clicon/clixon.git
```

1.5 1.5 Support

Clixon interaction is best done posting issues, pull requests, or joining the Matrix clixon forum https://matrix.to/#/#clixonforum:matrix.org.

1.6 1.6 Bug reports

Report bugs via Github issues

1.7 1.7 Reference docs

The user-manual is this document. For reference documentation of the C-code, Doxygen is used. To build the reference documentation you need to check out the source code, and type `make doc`, eg:

```
git clone git@github.com:clicon/clixon.git
cd clixon
./configure
make doc
```

direct your browser to:

```
file:///<your home path>/clixon/doc/html/index.html
```

CHAPTER

TWO

2 INSTALLATION

2.1 2.1 Ubuntu Linux

This section describes how to build Clixon from source on Ubuntu Linux. You can use this as a base for other platforms as well since many steps (such as prereqs) are similar.

Further, the *vagrant* scripts show how to build for some other Linux variants.

2.1.1 2.1.1 Prerequisites

General prerequisites

Install packages:

```
sudo apt-get install flex bison
```

Install and build CLIgen:

```
git clone https://github.com/clicon/cligen.git
cd cligen;
configure
make;
sudo make install
```

Add a clixon user and group, using useradd and usermod:

```
sudo useradd -M -U clixon
sudo usermod -a -G clixon <youruser>  # Remember to re-log in for this to take effect
sudo usermod -a -G clixon www-data # Only if RESTCONF
```

If you do not require RESTCONF, then continue with *Build clixon from source*.

RESTCONF HTTP Support

The RESTCONF implementation supports two HTTP configurations:

1. *Clixon native HTTP server*
2. *FastCGI for reverse proxy*

Clixon native HTTP server

Native http server requires openssl 1.1 or later.

Install TLS:

```
sudo apt-get install libssl-dev
```

Thereafter configure using default options:

```
configure
```

FastCGI for reverse proxy

FastCGI requires support for Nginx or similar reverse HTTP proxy:

```
sudo apt-get install nginx libfcgi-dev
```

Then, when building clixon from source (see below), configure clixon with:

```
configure --with-restconf=fcgi
```

Note that the libfcgi-dev package might not exist in Ubuntu 18 bionic or later, in which case need to build *fcgi* from source.

Note also that the 'fcgi' installation package might have a different name on other Linux distributions, such as "fcgi-dev" (alpine), "fcgi" (arch), "fcgi-devkit" (freebsd).

2.1.2 2.1.2 Build Clixon from source

Download clixon source code:

```
git clone https://github.com/clicon/clixon.git
```

Configure Clixon using one of the following RESTCONF configurations:

```
configure --with-restconf=native # clixon native HTTP server
configure --with-restconf=fcgi   # FastCGI support for reverse proxy, the default
                                 # when no '--with-restconf' option is specified
configure --without-restconf     # Do not build restconf
```

For more configure options see: *Configure options*.

Build and install:

```
make                    # Compile
sudo make install       # Install libs, binaries, config-files and include-files
sudo ldconfig           # To link new dynamic libraries
```

Building the example and utils

To build and install the example app, from the top level clixon directory:

```
make example
cd example
sudo make install
```

To build the utils for running the tests, from the top level clixon directory:

```
sudo apt install libcurl4-openssl-dev
cd util
make
sudo make install
```

See also the *Quickstart section* section for building a complete *hello world* example.

2.2 2.2 FreeBSD

FreeBSD has ports for both cligen and clixon available. You can install them as binary packages, or you can build them in a ports source tree locally.

If you install using binary packages or build from the ports collection, the installation locations comply with FreeBSD standards and you have some assurance that the installed package is correct and functional.

The Nginx setup for RESTCONF is altered - the system user www is used, and the restconf daemon is placed in /usr/local/sbin.

2.2.1 2.2.1 Binary package install

To install the pre-built binary package, use the FreeBSD pkg command:

```
% pkg install clixon
```

This will install clixon and all the dependencies needed.

2.2.2 2.2.2 Build from source on FreeBSD

If you prefer you can also build clixon from the FreeBSD ports collection

Once you have the Ports Collection installed, you build clixon like this

```
% cd /usr/ports/devel/clixon
% make && make install
```

One issue with using the Ports Collection is that it may not install the latest version from GitHub. The port is generally updated soon after an official release, but there is still a lag between it and the master branch. The maintainer for the port tries to assure that the master branch will compile always, but no FreeBSD specific functional testing is done.

2.3 2.3 Systemd

Once installed, Clixon can be setup using systemd. The following shows an example with the backend and restconf daemons from the main example. Install them as /etc/systemd/system/example.service and /etc/systemd/system/example_restconf.service, for example.

2.3.1 2.3.1 Systemd backend

The backend service is installed at /etc/systemd/system/example.service, for example. Note that in this example, the backend installation requires the restconf service, which is not necessary.

```
[Unit]
Description=Starts and stops a clixon example service on this system
Wants=example_restconf.service
[Service]
Type=forking
User=root
RestartSec=60
Restart=on-failure
ExecStart=/usr/local/sbin/clixon_backend -s running -f /usr/local/etc/example.xml
[Install]
WantedBy=multi-user.target
```

2.3.2 2.3.2 Systemd restconf

The Restconf service can be installed at, for example, /etc/systemd/system/example_restconf.service:

```
[Unit]
Description=Starts and stops an example clixon restconf service on this system
Wants=example.service
After=example.service
[Service]
Type=simple
User=root
Restart=on-failure
ExecStart=/usr/local/sbin/clixon_restconf -f /usr/local/etc/example.xml
[Install]
WantedBy=multi-user.target
```

The restconf daemon can also be started internally using the clixon-lib process-control RPC. For more info, see *Restconf section*.

2.4 2.4 Docker

Clixon can run in a docker container. As an example the *docker* directory has boilerplate code for building Clixon in a container:

```
cd docker/base
make docker
```

For complete examples see:

- Hello world
- Clixon CI test container
- Openconfig

2.5 2.5 Vagrant

Clixon uses vagrant in testing. For example to start a Freebsd vagrant host, install Clixon and run the test suite, do

```
cd test/vagrant
./vagrant.sh generic/freebsd12
```

Other platforms include: ubuntu/bionic64 and generic/centos8. To look at how Clixon is installed natively on those platforms please look in the build scripts under test/vagrant/.

2.6 2.6 OpenWRT

See Clixon cross-compiler for Openwrt

2.7 2.7 Prereqs from source

2.7.1 2.7.1 FCGI

For RESTCONF using fcgi build fcgi from source as follows:

```
git clone https://github.com/FastCGI-Archives/fcgi2
cd fcgi2
./autogen.sh
./configure --prefix=/usr
make
sudo make install
```

2.8 2.8 SSH subsystem

You can expose `clixon_netconf` as an SSH subsystem according to *RFC 6242*. Register the subsystem in `/etc/sshd_config`:

```
Subsystem netconf /usr/local/bin/clixon_netconf
```

and then invoke it from a client using:

```
ssh -s <host> netconf
```

2.9 2.9 Configure options

The Clixon *configure* script (generated by autoconf) includes several options apart from the standard ones.

These include (standard options are omitted)

--enable-debug Build with debug symbols, default: no

--enable-yang-patch Enable RFC 8072 YANG patch (plain patch is always enabled)

--enable-publish Enable publish of notification streams using SSE and curl

--disable-http1 Disable native http/1.1 (ie http/2 only)

--disable-nghttp2 Disable native http/2 using libnghttp2 (ie http/1 only)

--with-cligen=dir Use CLIGEN here

--with-restconf=native RESTCONF using native http. (DEFAULT)

--with-restconf=fcgi RESTCONF using fcgi/ reverse proxy.

--without-restconf No RESTCONF

--with-configfile=FILE Set default path to config file

--with-libxml2 Use gnome/libxml2 regex engine

--without-sigaction Disable sigaction logic (some platforms do not support SA_RESTART mode)

--with-yang-installdir=DIR Install Clixon yang files here (default: ${prefix}/share/clixon)

--with-yang-standard-dir=DIR Location of standard IETF/IEEE YANG specs for tests and example (default: $prefix/share/yang/standard). You can retrieve the standard files at https://github.com/YangModels/yang

--with-clicon-user=user Run as this user in example and test

--with-clicon-group=group Run as this group in example and test

There are also some variables that can be set, such as:

```
./configure LINKAGE=static                    # Build static libraries
./configure CFLAGS="-O1 -Wall" INSTALLFLAGS="" # Use other CFLAGS
```

Note, you need to reconfigure and recompile from scratch if you want to build static libs

2.10 2.10 macOS

Clixon can be built on macOS, however not all tests will pass at this moment and there might be pieces which will not run properly.

A few packages must be installed using for example HomeBrew:

```
brew install openssl nghttp2
```

Since we install a few libraries from HomeBrew we might want to set C and library paths:

```
$ export LIBRARY_PATH=$LIBRARY_PATH:/opt/homebrew/opt/openssl/lib
$ export C_INCLUDE_PATH=/opt/homebrew/opt/openssl/include/
```

Then Cligen and Clixon can be built as normal. Since Clixon will install things in "/usr/local/sbin/" you might want to add this to PATH. Either temporarily using:

```
export PATH=$PATH:/usr/local/sbin/
```

Or permanently by adding the above to .bash_profile or similar.

Since macOS don't use systemd or similar you'll have to start and stop clixon_backend etc manually.

CHAPTER

THREE

3 QUICK START

This section describes how to run the *hello world* example available in source code at: clixon hello example.

Clixon is not a system in itself, it is a support system for an application. In this case, the "application" is hello world. The hello world application is very simple where the application semantics is completely described by a YANG and CLI specification.

A more advanced application have backend and frontend plugins that define application-specific semantics. No plugins are present in the hello world application.

The hello world example can be run both natively on the host and in a docker container.

3.1 3.1 Host native

3.1.1 3.1.1 Clixon

Go through the *install instructions* to install Clixon on your platform. This includes installing CLIgen, Clixon, creating users, groups, etc.

In short:

```
git clone https://github.com/clicon/cligen.git
cd cligen
./configure
make && sudo make install
git clone https://github.com/clicon/clixon.git
cd clixon
./configure
make && sudo make install
```

Then proceed with host application install.

3.1.2 3.1.2 Files

Files relevant to the hello world example are:

- hello.xml: the XML configuration file
- clixon-hello@2019-04-17.yang: the YANG spec
- hello_cli.cli: the CLIgen spec
- startup_db: The startup datastore containing restconf port configuration
- Makefile: install of specs, normally compile of plugins

3.1.3 3.1.3 Install and run

Checkout and configure the examples on the top-level:

```
git clone https://github.com/clicon/clixon-examples.git
cd clixon-examples
./configure
```

Compile and install:

```
cd hello/src
make && sudo make install
```

Start backend in the background:

```
sudo clixon_backend
```

Start cli:

```
clixon_cli
```

3.2 3.2 Using the CLI

The example CLI allows you to modify and view the data model using *set*, *delete* and *show* via generated code.

The following example shows how to add a very simple configuration *hello world* using the generated CLI. The config is added to the candidate database, shown, committed to running, and then deleted.

```
olof@vandal> clixon_cli
cli> set <?>
  hello
cli> set hello world
cli> show configuration
hello world;
cli> commit
cli> delete <?>
  all                   Delete whole candidate configuration
  hello
cli> delete hello
cli> show configuration
```

(continues on next page)

(continued from previous page)

```
cli> commit
cli> quit
olof@vandal>
```

3.3 3.3 Netconf

Clixon also provides a Netconf interface. The following example starts a netconf client form the shell vi stdio, adds the hello world config, commits it, and shows it:

```
olof@vandal> clixon_netconf -q
<?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:1.1</capability></capabilities></hello>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><edit-config><target><candidate/></
↪target><config><hello xmlns="urn:example:hello"><world/></hello></config></edit-config>
↪</rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><ok/></rpc-reply>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><commit/></rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><ok/></rpc-reply>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><get-config><source><running/></
↪source></get-config></rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><data><hello xmlns=
↪"urn:example:hello"><world/></hello></data></rpc-reply>]]>]]>
olof@vandal>
```

3.4 3.4 Restconf

By default, Clixon uses *Native http*: supporting http/1 and http/2 (libnghttp2). The http server is integrated with the clixon restconf daemon and needs no extra installations, apart from ensuring you have server and client certs for https.

As an alternative, you can use the *FCGI* solution, where instead a reverse proxy such as Nginx uses an internal FCGI socket communication to communicate with Clixon. A reverse proxy, such as NGINX, needs to be configured. For more info about the fcgi solution, see *Restconf section*.

3.4.1 3.4.1 Start and run

Regardless of which RESTCONF variant is used, start the restconf daemon as follows:

```
sudo clixon_restconf
```

Start sending restconf commands (using Curl):

```
olof@vandal> curl -X POST http://localhost/restconf/data -d '{"clixon-hello:hello":{
↪"world":null}}'
olof@vandal> curl -X GET http://localhost/restconf/data
{
   "data": {
     "clixon-hello:hello": {
```

(continues on next page)

(continued from previous page)

```
      "world": null
    }
  }
}
```

3.5 3.5 Docker container

You can run the hello example as a pre-built docker container, on a *x86_64* Linux. See instructions in the clixon docker hello example.

First, the container is started with the backend running:

```
$ sudo docker run --rm -p 8080:80 --name hello -d clixon/hello
```

Then a CLI is started

```
$ sudo docker exec -it hello clixon_cli
cli> set ?
 hello
cli> set hello world
cli> show configuration
hello world;
```

Or Netconf:

```
$ sudo docker exec -it clixon/clixon clixon_netconf
<?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:1.1</capability></capabilities></hello>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><get-config><source><candidate/></
↪source></get-config></rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><data/></rpc-reply>]]>]]>
```

Or using restconf using curl on exposed port 8080:

```
$ curl -X GET http://localhost:8080/restconf/data/hello:system
```

3.6 3.6 Next steps

The hello world example only has a Yang spec and a template CLI spec. For more advanced applications, customized backend, CLI, netconf and restconf code callbacks becomes necessary.

Further, you may want to add upgrade, RPC:s, state data, notification streams, authentication and authorization. The main example contains such capabilities.

CHAPTER

FOUR

4 STANDARDS

4.1 4.1 YANG

YANG and XML are central to Clixon. Yang modules are used as a specification for encoding XML or JSON configuration and state data. The YANG spec is also used to generate an interactive CLI, NETCONF and RESTCONF clients, as well as the format of the XML datastore.

The YANG standards that Clixon follows include (see also *netconf*):

- YANG 1.0 RFC 6020
- YANG 1.1 RFC 7950
- YANG library RFC 8525 (partly)

Clixon deviates from the YANG standard as follows (reference to RFC7950 sections in parenthesis):

Not implemented:

- augment in a uses sub-clause (7.17) (module-level augment is implemented)
- instance-identifier type (9.13)
- status (7.21.2)
- YIN (13)
- Default values on leaf-lists (7.7.2)
- error-message is not implemented as sub-statement of "range", "length" and "pattern"

Further:

Clixon supports the following extended XPath functions (10):

- current()
- deref()
- derived-from(),
- derived-from-or-self()
- bit-is-set()
- The following extended XPath functions are *not* supported (10):
  - re-match()
  - enum-value()

See also support of standard XPath functions *XML and XPath*

4.1.1 4.1.1 Regular expressions

Clixon supports two regular expression engines:

Posix
: The default method, The regexps:s are translated to posix before matching with the standard Linux regex engine. This translation is not complete but can be considered "good-enough" for most yang use-cases. For reference, all standard Yang models have been tested.

Libxml2
: Libxml2 uses the XSD regex engine. This is a complete XSD engine but you need to compile and link with libxml2 which may add overhead.

To use libxml2 in clixon you need enable libxml2 in both cligen and clixon:

```
./configure --with-libxml2 # both cligen and clixon
```

You then need to set the following configure option:

```
<CLICON_YANG_REGEXP>libxml2</CLICON_YANG_REGEXP>
```

4.1.2 4.1.2 Metadata

Clixon implements Defining and Using Metadata with YANG RFC 7952 for XML and JSON.

This means that Yang-derived meta-data defined with:

```
md:annotation <name>
```

is defined for attributes so that they can be mapped from XML to JSON, for example.

Assigned meta-data are hardcoded. The following attributes are defined:

- ietf-netconf-with-defaults:default from RFC 6243 / RFC 8040

4.1.3 4.1.3 Schema mount

Yang schema mount is supported as defined in: RFC 8528: YANG Schema Mount .

Enable by enable the *CLICON_YANG_SCHEMA_MOUNT* configuration option.

4.2 4.2 NETCONF

Clixon implements the following NETCONF RFC:s:

- RFC 5277: NETCONF Event Notifications
- RFC 6022: YANG Module for NETCONF Monitoring.
- RFC 6241: NETCONF Configuration Protocol
- RFC 6242: Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 6243 With-defaults Capability for NETCONF * RFC 8071: NETCONF Call Home and RESTCONF Call Home. NETCONF over SSH (external) and RESTCONF call home (internal) over TLS are implemented.
- RFC 8341: Network Configuration Access Control Model (NACM). Notification not implemented.

The following RFC6241 capabilities/features are hardcoded in Clixon:

- :candidate (RFC6241 8.3)
- :validate (RFC6241 8.6)
- :xpath (RFC6241 8.9)
- :notification (RFC5277)
- :with-defaults (RFC6243)

The following features are optional and can be enabled by setting CLICON_FEATURE:

- :confirmed-commit:1.1 (RFC6241 8.4)
- :startup (RFC6241 8.7)
- :writable-running (RFC6241 8.2) - just write to running, no commit semantics

Clixon does *not* support the following NETCONF features:

- :url capability
- copy-config source config
- edit-config testopts
- edit-config erropts
- edit-config config-text
- edit-config operation

Further, in *get-config* filter expressions, the RFC6241 XPath Capability is preferred over default subtrees. This has two reasons:

1. XPath has better performance since the underlying system uses xpath, and subtree filtering is done after the complete tree is retrieved.
2. Subtree filtering does not support namespaces yet.

Clixon supports netconf locks in default settings but *not* if `CLICON_DATASTORE_CACHE` is `nocache` mode.

4.2.1 4.2.1 RFC 6022

Clixon extends the RFC 6022 session parameter `transport` with "cli", "restconf", "netconf" and "snmp". In particular, the `clixon_netconf` application uses stdio to get input and print output and is used in a "piping" fashion, for example directly in a terminal shell or as a part of a SSH sub-system, and therefore has no direct knowledge of whether the NETCONF transport is over SSH or not.

The `source-host` parameter is set only in certain circumstances when the source host is in fact known. This includes native RESTCONF for example.

Further, `hello` counters are backend based, ie the internal protocol, which means hellos from RESTCONF, SNMP and CLI clients are included and that eventual dropped hello messages from external NETCONF sessions are not.

4.2.2 4.2.2 Default handling

Clixon treats default data according to what is defined as *explicit basic mode* in RFC 6243: With-defaults Capability for NETCONF, i.e. the server consider any data node that is not explicitly set data to be default data.

One effect is that if you view the contents of datastores (or import/export them), they should be in *explicit basic mode*.

The *:with-defaults* capability indicates that clixon default behaviour is explicit and also indicates that additional retrieval modes supported by the server are:.

- explicit
- trim
- report-all
- report-all-tagged

Internally in memory, however, *report-all* is used.

4.3 4.3 RESTCONF

Clixon supports the two RESTCONF compile-time variants: *FCGI* and *Native*. Both implements RFC 8040: RESTCONF Protocol.

The following features of RFC8040 are supported:

- OPTIONS, HEAD, GET, POST, PUT, DELETE, PATCH
- Stream notifications (Sec 6)
- Query parameters: *insert*, *point*, *content*, *depth*, *start-time*, *stop-time* and *with-defaults*.
- Monitoring (Sec 9)

The following features are *not* implemented:

- ETag/Last-Modified
- Query parameters: *fields* and *filter*

RESTCONF event notification as described in RFC7950 section 6 is supported as follows:

- is *not* supported by *native*
- is supported by *FCGI*

NMDA is partly supported according to RFC 8324 and RFC 8527. With-defaults and with-origin are not implemented.

RFC 8072: YANG Patch Media Type is not implemented.

In the native mode, Clixon also supports:

- HTTP/1.1 as transport using a native implementation (RFC 7230),
- HTTP/2 as transport implemented by libnghttp2 (RFC7540),
- Transport Layer Security (TLS) implemented by libopenssl, versions 1.1.1 and 3.0
- ALPN as defined in RFC 7301 for http/1, http/2 protocol selection by libopenssl

4.4 4.4 SNMP

The Clixon-SNMP frontend implements the MIB-YANG mapping as defined in RFC 6643.

4.5 4.5 XML and XPath

Clixon has its own implementation of XML and XPath. See more in the detailed API reference.

The XML-related standards include:

- XML 1.0. (DOCTYPE/ DTD not supported)
- Namespaces in XML 1.0
- XPath 1.0

Clixon XML supports version and UTF-8 only.

The following XPath axes are supported:

- child,
- descendant,
- descendant-or-self,
- self
- parent

The following xpath axes are *not* supported: preceding, preceding_sibling, namespace, following_sibling, following, ancestor,ancestor_or_self, and attribute

The following XPath functions as defined in Section 2.3 / 4 of the XPath 1.0 standard are supported:

- contains()
- count()
- false()
- name()
- node()
- boolean()
- not()
- position()
- text()
- true()

The following standard XPath functions are *not* supported:

- ceiling
- comment
- concat
- floor
- id

- lang
- last
- local-name
- namespace-uri
- normalize-space
- number
- processing-instructions
- round
- starts-with
- string
- substring
- substring-after
- substring-before
- sum
- translate

4.6 4.6 Pagination

The pagination solution is based on the following drafts:

- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-00.html
- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-nc-00.html
- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-rc-00.html

See *Pagination section* for more info.

4.7 4.7 Unicode

Unicode is not supported in YANG and XML.

4.8 4.8 JSON

Clixon implements JSON according to:

- ECMA JSON Data Interchange Syntax
- RFC 7951 JSON Encoding of Data Modeled with YANG.
- RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format

CHAPTER

FIVE

5 CONFIGURATION

Clixon configuration files are encoded in XML and modeled by YANG. By default, the main config file is installed as `/usr/local/etc/clixon.xml`, but can be changed by the `-f <file>` command-line option.

The YANG specification for Clixon configuration is clixon-config.yang. This configuration file is updated regularly.

Normally, all Clixon processes (backend, cli, netconf, restconf) use the same configuration, although some options are not valid for all processes. You can however have different configuration files for different clients by using the `-f` option.

Loading an obsolete config option will result in an error.

Please consult the clixon-config YANG spec directly if you want detailed description of config options.

5.1 5.1 Example

The following is the configuration file of a simple example:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
  <CLICON_CONFIGDIR>/usr/local/etc/clixon.d</CLICON_CONFIGDIR>
  <CLICON_FEATURE>*:*</CLICON_FEATURE>
  <CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
  <CLICON_YANG_MODULE_MAIN>clixon-hello</CLICON_YANG_MODULE_MAIN>
  <CLICON_CLI_MODE>hello</CLICON_CLI_MODE>
  <CLICON_CLISPEC_DIR>/usr/local/lib/hello/clispec</CLICON_CLISPEC_DIR>
  <CLICON_SOCK>/usr/local/var/hello.sock</CLICON_SOCK>
  <CLICON_BACKEND_PIDFILE>/usr/local/var/hello.pidfile</CLICON_BACKEND_PIDFILE>
  <CLICON_XMLDB_DIR>/usr/local/var/hello</CLICON_XMLDB_DIR>
  <CLICON_STARTUP_MODE>init</CLICON_STARTUP_MODE>
  <restconf>
     <enable>true</enable>
  </restconf>
</clixon-config>
```

The option `CLICON_CONFIGFILE` is special, it must be available before the configuration file is found (see *Loading the configuration*), which means that the value in the file is a no-op.

The `restconf` clause defines RESTCONF configuration options as described in the *restconf section* section.

5.2 5.2 Loading the configuration

Clixon finds its configuration files, according to the following method:

1. Start a clixon program with the `-f <FILE>` option. For example:

```
clixon_backend -f FILE
```

2. At install time, Use the `--with-configfile=FILE` option to configure a default location:

```
./configure --with-configfile=FILE
```

3. At install time: `./configure --with-sysconfig=<dir>` when configuring. Then FILE is `<dir>/clixon.xml`
4. At install time: `./configure --sysconfig=<dir>` when configuring. Then FILE is `<dir>/etc/clixon.xml`
5. If none of the above: FILE is `/usr/local/etc/clixon.xml`

The following options control the Clixon configuration:

CLICON_CONFIGFILE
: The configure file itself. Due to bootstrapping reasons, its value is meaningless in a file but can be useful for documentation purposes.

CLICON_CONFIGDIR
: A directory of extra configuration files loaded after the main configuration. It can also be specified using the `-E <dir>` command-line option. These extra configuration files are read in alphabetical order after the main configuration as follows:

  - leaf values are overwritten
  - leaf-list values are appended

5.3 5.3 Runtime modification

You can modify clixon options at runtime by using the `-o` option to modify the configuration specified in the configuration file. For example, add `usr/local/share/ietf` to the list of directories where yang files are searched for:

```
clixon_cli -o CLICON_YANG_DIR=/usr/local/share/ietf
```

5.4 5.4 Features

`CLICON_FEATURE` is a list of values, describing how Clixon supports features.

A value is specified as one of the following:

- `<module>:<feature>` : enable a specific feature in a specific module
- `*:*` : enable all features in all modules
- `<module>:*` : enable all features in the specified module
- `*:<feature>` : enable the specific feature in all modules.

Example:

```
<CLICON_FEATURE>ietf-netconf:startup</CLICON_FEATURE>
<CLICON_FEATURE>ietf-netconf:*</CLICON_FEATURE>
<CLICON_FEATURE>*:*</CLICON_FEATURE>
```

Supplying the `-o` option adds a value to the feature list.

Clixon have three hardcoded features:

- ietf-netconf:candidate (RFC6241 8.3)
- ietf-netconf:validate (RFC6241 8.6)
- ietf-netconf:xpath (RFC6241 8.9)

5.5 5.5 Finding YANG files

The example have two options for finding Yang files:

```
<CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
<CLICON_YANG_MODULE_MAIN>clixon-hello</CLICON_YANG_MODULE_MAIN>
```

which means that Yang files are searched for in `/usr/local/share/clixon` and that module `clixon-hello` is loaded. Note:

- `clixon-hello.yang` must be present in `/usr/local/share/clixon`
- Clixon itself may load several YANG files as part of the system startup, such as `clixon-config.yang`. These must all reside in the list of `CLICON_YANG_DIR`:s.
- When a Yang file is loaded, it may contain references to other Yang files (eg using `import` and `include`). They must also be found in the list of `CLICON_YANG_DIR`:s.

The following configuration file options control the loading of Yang files:

CLICON_YANG_DIR
: A list of directories (yang dir path) where Clixon searches for module and submodules *recursively*.

CLICON_YANG_MAIN_FILE
: Load a specific Yang module given by a file.

CLICON_YANG_MODULE_MAIN
: Specifies a single module to load. The module is searched for in the yang dir path.

CLICON_YANG_MODULE_REVISION
: Specifies a revision to the main module.

CLICON_YANG_MAIN_DIR
: Load all yang modules in this directory, not recursively.

Note that the special `YANG_INSTALLDIR` autoconf configure option, by default `/usr/local/share/clixon` should be included in the yang dir path for Clixon system files to be found.

You can combine the options, however, if there are different variants of the same module, more specific options override less specific. The precedence of the options are as follows:

1. `CLICON_YANG_MAIN_FILE`
2. `CLICON_YANG_MODULE_MAIN`
3. `CLICON_YANG_MAIN_DIR`

Note that using CLICON_YANG_MAIN_DIR Clixon may find several files containing the same Yang module. Clixon will prefer the one without a revision date if such a file exists. If no file has a revision date, Clixon will prefer the newest.

5.6 5.6 Standard YANG files

The main examples and tests require IETF RFC standard YANGs. If you want to run the main example or run tests, you need to make them locally available by checking out `https://github.com/YangModels/yang` which has subdir `standard`. By default this directory is `/usr/local/share/yang` You can change this location by:

```
./configure --with-yang-standard-dir=DIR
```

Note that you do not need this for the clixon runtime.

5.7 5.7 Extending the configuration

You can extend the options with an application-specific YANG file where you augment the regular "clixon-config" as follows:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
  <CLICON_CONFIG_EXTEND>clixon-myconfig</CLICON_CONFIG_EXTEND>
  ...
```

You then install your own "clixon-myconfig.yang" where you add your own config options. Example:

```
module clixon-myconfig {
  yang-version 1.1;
  namespace "http://example.org/myconfig";
  ...
  import clixon-config {
      prefix "cc";
  }
  augment "/cc:clixon-config" {
      description
          "My extended options";
      leaf MYOPT {
          type string;
      }
```

You can now use your extended options in the regular config file, along with the basic ones, but with another namespace:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
  <CLICON_CONFIG_EXTEND>clixon-myconfig</CLICON_CONFIG_EXTEND>
  ...
  <MYOPT xmlns="http://example.org/myconfig">/usr/local/share/myopt</MYOPT>
```

You can also use the regular C-API to access the values of the options, eg:

```
char *val = clicon_option_str(h, "MYOPT");
```

CHAPTER

SIX

6 PLUGINS

Plugins are the "glue" that binds the underlying system to Clixon and where application semantics is added by an application developer.

The backend, CLI, Restconf and Netconf applications may use plugins, although the main use of plugins are the *Backend section* plugins.

Plugins are written in C as dynamically loaded modules (`.so` files). At startup, the application looks in the assigned directory and loads all files with `.so` suffixes from that dir in *alphabetical* order.

Plugins are located as follows:

- Backend plugins are are in `CLICON_BACKEND_DIR`
- CLI plugins are are in `CLICON_CLI_DIR`
- NETCONF plugins are in `CLICON_NETCONF_DIR`
- RESTCONF plugins are in `CLICON_CLI_DIR`;

For example, to load all backend plugins from: `/usr/local/lib/example/backend`:

```
<CLICON_BACKEND_DIR>/usr/local/lib/example/backend</CLICON_BACKEND_DIR>
```

Callbacks are registered in two ways:

- *clixon_plugin_init* : Return an API struct containing a fixed set of callbacks
- *register functions* : Register callback functions for more complex interaction

Further, CLI callbacks as described in *CLI section* are special in the way that they are invoked from CLIgen specifications, not from the application itself.

6.1 6.1 Clixon_plugin_init

On startup, the application loads each plugin and calls `clixon_plugin_init` in that plugin. The function is expected to return an API struct `clixon_plugin_api` defining a set of static callbacks. The init function may return `NULL` in which case it is logged and ignored.

Once the plugin is loaded, it awaits callbacks from the application.

An example of a minimal plugin is as follows:

```
static clixon_plugin_api api = {
  "example",                              /* Plugin name */
  clixon_plugin_init,                     /* init - must be called clixon_plugin_init */
```

(continues on next page)

(continued from previous page)

```
  example_start,                          /* start */
  example_exit,                           /* exit */
  example_extension,                      /* yang extensions */
  .ca_daemon=example_daemon,              /* Plugin daemonized () (backend only) */
};

clixon_plugin_api *
clixon_plugin_init(clixon_handle h)
{
   return &api;
}
```

First the `clixon_plugin_api` struct defines the callbacks. There are two blocks of callbacks:

- Common callbacks for backend, cli, netconf, and restconf
- Application-specific callbacks. In the example, only `ca_daemon` is specific to backends. By far, the backend have most specific callbacks.

Second, the `clixon_plugin_init()` function is defined and (at a minimum) returns the struct.

The following callbacks are common plugins for all Clixon applications:

init
: Clixon plugin init function, called immediately after plugin is loaded into the application. The name of the function must be called `clixon_plugin_init`. It returns a struct with the name of the plugin, and all other callback names.

start
: Called when application is started and initialization is complete, but before the application is placed in the background and privileges are dropped (if applicable)

exit
: Called just before plugin is unloaded at exit

extension
: Called at parsing of yang modules containing an extension statement. A plugin may identify the extension by its name, and perform actions on the yang statement, such as transforming the yang in-memory. A callback is made for every statement, which means that several calls per extension can be made. See *Misc section* and the main example on how to use extensions.

yang_mount
: Called when populating a RFC8525 mount-point. See *YANG section* for more info.

yang_patch
: Patch a yang module. This may be necessary with an imported faulty or non-compliant YANG module

errmsg
: Customize a netconf error message for CLI return, log or debug messages. See *Error section* and the main example.

version
: Print a plugin-specific version stringh

See *Backend section* for backend specific callbacks, as well as corresponding manual sections for netconf/restconf/cli callbacks.

6.2 6.2 Registered callbacks

A second group of callbacks use register functions. This is a more detailed mechanism than the fixed callbacks described previously, but are only defined to a limited sets of functions:

- `rpc_callback_register()` - for user-defined RPC callbacks. Applicable for NETCONF, RESTCONF and backend.
- `action_callback_register()` - for user-defined Action callbacks. Applicable for backend.
- `upgrade_callback_register()` - for upgrading, see *Upgrade section*. Applicable only for backend.
- `clixon_pagination_cb_register()` - for pagination, as described in *Pagination section*. Applicable only for backend.

A user may register may register a callback for an incoming RPC, and that function will be called.

There may be several callbacks for the same RPC. The order the callbacks are registered are as follows:

1. plugin_init
2. backend_rpc_init (where system callbacks are registered)
3. plugin_start

Which means if you register a copy-config callback in (1), it will be called *before* the system copy-config callback registered from (2) backend_rpc_init. If you register a copy-config in (3) plugin-start it will be called *after* the system copy-config.

Second, if there are more than one reply (eg `<rpc-reply/><rpc-reply/>`) only the first reply will be parsed and used by the cli/netconf/restconf clients.

If you want to take the original and modify it, you should therefore register the callback in plugin_start (3) so that your callback will be called after the system RPC. Then you should modify the original reply (not add a new reply).

6.2.1 6.2.1 Example: RPC callback

This example shows how to define a new RPC in YANG for the backend, register a callback function in C, read and write a parameter. It is revised slightly from the main example.

YANG:

```
module clixon-example {
  namespace "urn:example:clixon";
  ...
  rpc example {
      input {
          leaf x {
            type string;
            ...
      output {
          leaf y {
            type string;
            ...
```

Register RPC in clixon_plugin_init():

```
clixon_plugin_api *clixon_plugin_init(clixon_handle h)
{
   ...
   rpc_callback_register(h, example_rpc, NULL, "urn:example:clixon", "example");
```

Callback function reading value input x, modifying value and writing it as output value y:

```
static int
example_rpc(clixon_handle h,            /* Clicon handle */
            cxobj        *xe,           /* Request: <rpc><xn></rpc> */
            cbuf         *cbret,        /* Reply eg <rpc-reply>... */
            void         *arg,          /* client_entry */
            void         *regarg)       /* Argument given at register */
{
    char *val;
    val = xml_find_body(xe, "x");       /* Read x value of incoming rpc */
    cprintf(cbret, "<rpc-reply xmlns=\"%s\">", NETCONF_BASE_NAMESPACE);
    val[0]++;                           /* Increment first char */
    /* Construct reply */
    cprintf(cbret, "<y xmlns=\"urn:example:clixon\">%s</y>", val);
    cprintf(cbret, "</rpc-reply>");
```

Result netconf session:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="42">
   <example xmlns="urn:example:clixon">
      <x>42</x>
   </example>
</rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="42">
   <y xmlns="urn:example:clixon">52</y>
</rpc-reply>]]>]]>
```

6.2.2 6.2.2 Example: Action callback

This example follows RFC 7950 7.15.3.

An action is associated with a YANG node and can therefore not be registered at init, instead the start callback can be used, for example.

Register Action in example_reset():

```
int
example_start(clixon_handle h)
{
   ...
   if (action_callback_register(h, ya, example_action_reset, NULL) < 0)
     goto done;
```

6.3 6.3 Plugin callback guidelines

Note: This information is important to understand for the stability of clixon

The Clixon programs run as non-blocking *single-threaded* applications. It calls functions from within dynamically loaded modules. The callback code must be written with this programming model in mind. The behavior of the callback directly impacts the behavior of the caller and the whole system.

The most serious effect is when crash within a callback happens. This will cause the whole program to crash.

A more subtle problem is the environment of the program. Clixon will configure the environment, and it expects that the callback will return with the exact same environment intact. If you change a signal handler, a terminal configuration, etc. *you must restore the state as it was on entry*. Failure to do this can cause problems that are difficult to isolate and fix.

A list of things to watch out for (but not complete!):

- a crash in the plugin
- change of signal behaviour, such as blocking or assigning signal handlers
- change of terminal settings (for CLI callbacks)
- change of process privileges
- asynchronous calls
- If you fork or create threads, ensure the main program continues uninterrupted

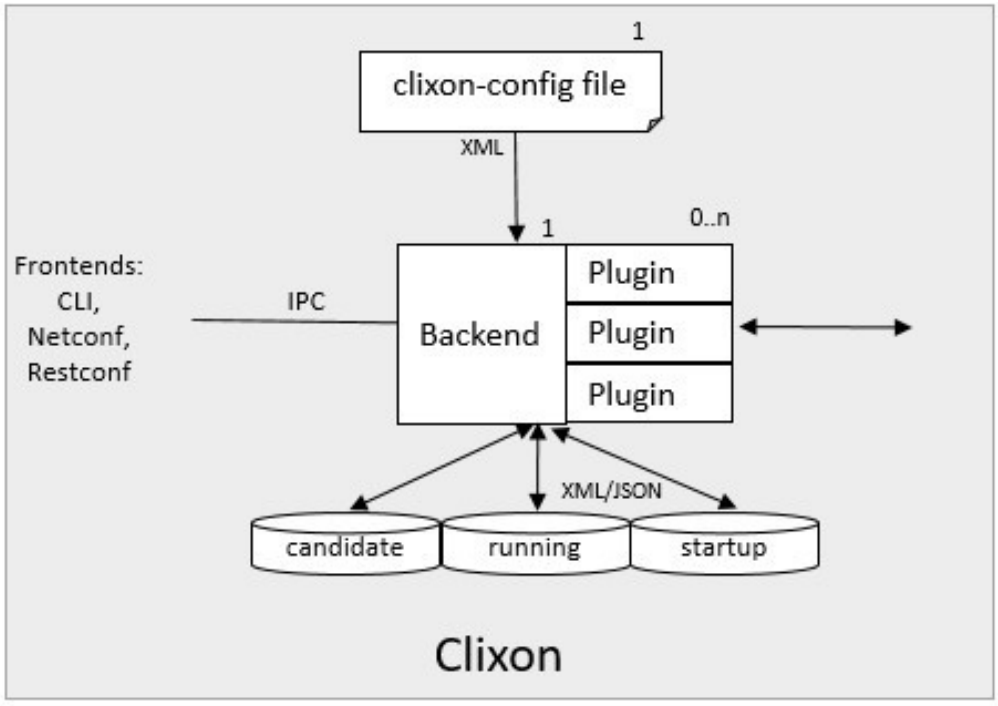
The following config option is related to checking callbacks:

CLICON_PLUGIN_CALLBACK_CHECK
: Enable check of resources before and after each callback. Checks are currently limited to signal and terminal settings

CHAPTER

SEVEN

7 BACKEND



The backend daemon is the central component in the Clixon architecture. It consists of a main module and a number of dynamically loaded plugins. The backend has four APIs:

configuration
: An XML file read at startup, possibly amended with *-o* options.

Internal interface / IPC
: A NETCONF socket to frontend clients. This is by default a UNIX domain socket but can also be an IPv4 or IPv6 TCP socket but with limited functionality.

Datastores
: XML (or JSON) files storing configuration. The three main datastores are *candidate*, *running* and *startup*. A user edits the candidate datastore, commits the changes to running which triggers callbacks in plugins.

Application
: Backend plugins configure the base system with application-specific APIs. These API:s depend on how the underlying system is configured, examples include configuration files or a socket.

Note that a user typically does not access the datastores directly, it is possible to read, but write operations should not be done, since the backend daemon in most cases uses a datastore cache.

7.1 7.1 Command-line options

The backend have the following command-line options:

-h	Help
-D <level>	Debug level
-f <file>	Clixon config file
-E <dir>	Extra configuration directory
-l <option>	Log on (s)yslog, std(e)rr, std(o)ut, (n)one or (f)ile. Syslog is default. If foreground, then syslog and stderr is default.
-C <format>	Dump configuration options on stdout after loading. Format is one of xml|json|text
-d <dir>	Specify backend plugin directory
-p <dir>	Add Yang directory path (see CLICON_YANG_DIR)
-b <dir>	Specify datastore directory
-F	Run in foreground, do not run as daemon
-z	Kill other config daemon and exit
-a <family>	Internal backend socket family: UNIX|IPv4|IPv6
-u <path|addr>	Internal socket domain path or IP addr (see -a)
-P <file>	Process ID filename
-1	Run once and then quit (do not wait for events)
-s <mode>	Specify backend startup mode: none|startup|running|init)
-c <file>	Load extra XML configuration file, but do not commit.
-q	Quit startup directly after upgrading and print result on stdout
-U <user>	Run backend daemon as this user AND drop privileges permanently
-g <group>	Client membership required to this group (default: clicon)
-y <file>	Load yang spec file (override yang main module)
-o <option=value>	Give configuration option overriding config file (see clixon-config.yang)

7.1.1 7.1.1 Logging and debugging

In case of debugging, the backend can be run in the foreground and with debug flags:

```
clixon_backend -FD 1
```

Note that debug levels can be combined as described in Section *debugging*.

Logging is by default on syslog. Alternatively, logging can be made on a file using the *-l* option:

```
clixon_backend -lf<file>
```

When run in foreground, logging is by default done on both syslog and stderr.

In a debugging mode, it can be useful to run in *once-only* mode, where the backend quits directly after starting up, instead of waiting for events:

```
clixon_backend -F1D 1
```

It may be useful to see all config options after load, taking into account default values, config-dirs and option overriding. This is normally dumped in debug level 1.

But you can also make an explicit dump of all config options on stdout using the *-C* option:

```
clixon_backend -1C xml
```

7.2 7.2 Startup

The backend can perform startup in four different modes. The difference is how the running state is handled, i.e., what state the system is in when you start the daemon and how loading the configuration affects it:

none
: Do not touch running state. Typically after crash when running state and db are synched.

init
: Initialize running state. Start with a completely clean running state.

running
: Commit running db configuration into running state. Typically after reboot if a persistent running db exists.

startup
: Commit startup configuration into running state. After reboot when no persistent running db exists.

Use the `-s` option to select startup mode, example:

```
clixon_backend -s running
```

You may also add a default method in the configuration file:

```
<clixon-config xmlns="http://clicon.org/config">
  ...
  <CLICON_STARTUP_MODE>init</CLICON_STARTUP_MODE
</clixon-config>
```

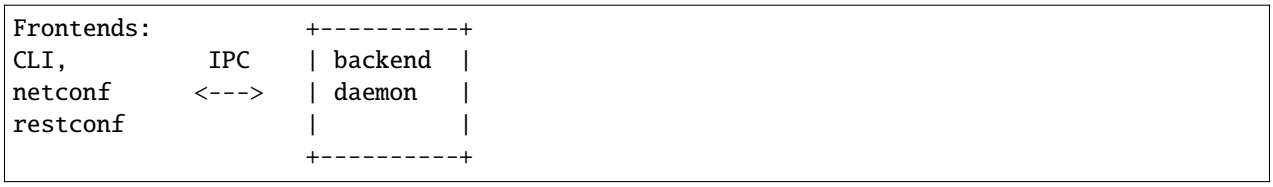
When loading the startup/tmp configuration, the following actions are performed by the system:

- Check syntax errors,
- Upgrade callbacks.
- Validation of the XML against the current Yang models
- If errors are detected, enter *failsafe* mode.

The following config option is related to startup:

CLICON_BACKEND_RESTCONF_PROCESS
: Enable process-control of restconf daemon, ie start/stop restconf daemon internally using fork/exec. Disable if you start the restconf daemon by other means.

7.3 7.3 IPC Socket



The Clixon backend creates a socket that the frontend clients can connect to. Communication is made over this IPC socket using internal Netconf. The following config options are related to the internal socket:

CLICON_SOCK_FAMILY
Address family for communicating with clixon_backend. One of: UNIX, IPv4, or IPv6. Can also be set with *-a* command-line option. Default is `UNIX` which denotes a UNIX domain socket.

CLICON_SOCK
If the address family of the socket is `AF_UNIX`: Unix socket for communicating with clixon_backend. If the family is `AF_INET` it denottes the IPv4 address;

CLICON_SOCK_PORT
Inet socket port for communicating with clixon_backend (only IPv4|IPv6). Default is port *4535*.

CLICON_SOCK_GROUP
Group membership to access clixon_backend UNIX socket. Default is *clicon*. This is not available for IP sockets.

7.4 7.4 Backend files

The following config options control files related to the backend:

CLICON_BACKEND_DIR
Location of backend `.so` plugins. Load all `.so` plugins in this dir as backend plugins in alphabetical order

CLICON_BACKEND_REGEXP
Regexp of matching backend plugins in CLICON_BACKEND_DIR. default: `*.so`

CLICON_BACKEND_PIDFILE
Process-id file of backend daemon

7.5 7.5 Backend plugins

This section describes backend-specific plugins, see *Plugin section* for a general description of plugins.

7.5.1 7.5.1 Clixon_plugin_init

Apart from the generic plugin callbacks (init, start, etc), the following callbacks are specific to the backend:

pre_daemon
Called just before server daemonizes(forks). Not called if in foreground.

daemon
Called after the server has daemonized and before privileges are dropped.

statedata
Provide state data XML from a plugin

reset
Reset system status

upgrade
General-purpose upgrade called once when loading the startup datastore

trans_{begin,validate,complete,commit,commit_done,revert,end,abort}
Transaction callbacks are invoked for two reasons: validation requests or commits. These callbacks are further described in *transactions* section.

7.5.2 7.5.2 Registered callbacks

The callback also supports three forms of registered callbacks:

- `rpc_callback_register()` - for user-defined RPC callbacks, see *Plugin section*.
- `action_callback_register()` - for user-defined Action callbacks.
- `upgrade_callback_register()` - for upgrading, see *Upgrade section*.
- `clixon_pagination_cb_register()` - for pagination, as described in *Pagination section*.

7.6 7.6 Transactions

Clixon follows NETCONF in its validate and commit semantics. Using the CLI or another frontend, you edit the *candidate* configuration, which is first *validated* for consistency and then *committed* to the *running* configuration.

A clixon developer writes commit functions to incrementally update a system state based on configuration changes. Writing transsaction callbacks is a core function of the clixon system.

The NETCONF validation and commit operation is implemented in clixon by a transaction mechanism, which ensures that user-written plugin callbacks are invoked atomically and revert on error.

Note: Clixon currently runs in a single thread and all transactions run from *begin,* to *end* without interruption. However, it is strongly recommended that the callbaks be thread-safe for future compatibility.

Warning: Clixon can only guarantee a consistent system state if you follow the guidelines below. Behavior that is not documented here is not guaranteed and subject to change without notice.

The phases of a transaction are:

begin
The begin callback indicates that a transaction is starting, but that the system level validation has not begun yet. Not much has happened at this point, and you should not rely on the transaction data or the XML ADD/DEL/CHANGE flags.

The intent of this callback is to notify the plugin that a transaction is beginning, and that it should allocate any resources necessary to handle the coming transaction.

validate
The validate callback is triggered by a client requesting a validation of the current change set. Before this callback, clixon has completed the system level YANG validation. All transaction data is valid at this point (including XML flags). In this callback, the plugin further validates the transaction data and returns a success/failure indication. It must not change the state of the system in any way. Successful completion should guarantee that the commit will

be without error. See the commit information below on why it is important to try. If a failure status is returned to clixon then the transaction is cancelled, and the transaction abort callback will be called from clixon.

You should place as much of the validation in the YANG model specification as possible. This does two things; it allows the user of the YANG model to understand the constraints clearly, and it is more efficient when the YANG validation catches problems before a transaction cycle happens.

complete
This callback is used to indicate to the plugin that the validation was successful and indicates that the commit is coming next. The transaction data will be valid when this callback is called. The complete callback must not change the state of the system.

The usefulness of this callback is debatable, and it is possible that this callback will be removed in a future version.

commit
This callback is called when the client requests a commit. The transaction data is valid, and both the system and plugin validations of the transaction data has completed without error.

The callback should perform the actions required to change the system state as indicated by the transaction target database.

Ideally all possible errors should be detected in the validate callback, but this clearly is a dream. A commit failure is a major event and leaves the system in an inconsistent state. In the event of an error, the callback must return an error status. Clixon will initiate its error processing to roll back the system to its state prior to the beginning of the transaction.

commit_done
Transaction when commit done. After this, the source db is copied to target and default values removed.

end
The end callback is called when a transaction has successfully completed.

You should not depend upon transaction data being valid at this point and no changes to the system state may occur (changes at this point prevent the possibility of error handling).

Any resources allocated in the begin callback should be released at this point. When the callback returns, the transaction is complete.

revert
When the revert callback is called the plugin must revert the system to the state prior (atomicity) to the beginning of the transaction. The transaction data is valid and is the same as in the commit callback. The code in the revert callback must assure that the system state matches the source database before returning.

A revert callback will be followed by an abort callback.

abort
If a validate or commit operation fails, the terminal action is to call the abort callback rather than the end callback. The common purpose of this callback is to release any resources allocated in the begin callback.

If an abort callback occurs after a commit, then the revert callback will be called prior to the abort. The plugin developer may do the actual reversion of the commit in the revert or abort callback, or split the duties as desired, but upon completion of the abort callback, it must be guaranteed that the system state will be as before the begin callback.

7.6.1 7.6.1 Transaction Examples

In a system with two plugins, for example, a transaction sequence looks like the following:

```
Backend   Plugin1    Plugin2
|          |          |
+--------->+--------->+ begin
|          |          |
+--------->+--------->+ validate
|          |          |
+--------->+--------->+ complete
|          |          |
+--------->+--------->+ commit
|          |          |
+--------->+--------->+ commit_done
|          |          |
+--------->+--------->+ end
```

If an error occurs in the commit call of plugin2, for example, the transaction is aborted and the commit reverted:

```
Backend   Plugin1    Plugin2
|          |          |
+--------->+--------->+ begin
|          |          |
+--------->+--------->+ validate
|          |          |
+--------->+---->X    + commit error
|          |          |
+--------->+          + revert
|          |          |
+--------->+--------->+ abort
```

7.6.2 7.6.2 Callbacks

In the the context of a callback, there are two XML trees:

src
: The original XML tree, such as "running"

target
: The new XML tree, such as "candidate"

There are three vectors pointing into the XML trees:

delete
: The delete vector consists of XML nodes in "src" that are removed in "target"

add
: The add vector consists of nodes that exists in "target" and do not exist in "src"

change
: The change vector consists of nodes that exists in both "src" and "target" but are different

All transaction callbacks are called with a transaction-data argument (td). The transaction data describes a system transition from a src to target state. The struct contains source and target XML tree (e.g. candidate/running) in the form of XML tree vectors (dvec, avec, cvec) and associated lengths. These contain the difference between src and

target XML, ie "what has changed". It is up to the validate callbacks to ensure that these changes are OK. It is up to the commit callbacks to enforce these changes in the configuration of the system.

The "td" parameter can be accessed with the following functions:

`uint64_t transaction_id(transaction_data td)`
Get the unique transaction-id

`void *transaction_arg(transaction_data td)`
Get plugin/application specific callback argument

`int transaction_arg_set(transaction_data td, void *arg)`
Set callback argument

`cxobj *transaction_src(transaction_data td)`
Get source database xml tree

`cxobj *transaction_target(transaction_data td)`
Get target database xml tree

`cxobj **transaction_dvec(transaction_data td)`
Get vector of xml nodes that are deleted src->target

`size_t transaction_dlen(transaction_data td)`
Get length of delete xml vector

`cxobj **transaction_avec(transaction_data td)`
Get vector of xml nodes that are added src->target

`size_t transaction_alen(transaction_data td)`
Get length of add xml vector

`cxobj **transaction_scvec(transaction_data td)`
Get vector of xml nodes that changed

`cxobj **transaction_tcvec(transaction_data td)`
Get vector of xml nodes that changed

`size_t transaction_clen(transaction_data td)`
Get length of changed xml vector

Flags

A programmer can also use XML flags that are set in "src" and "target" XML trees to identify what has changed. The following flags are used to makr the trees:

XML_FLAG_DEL
All deleted XML nodes in "src" and all its descendants

XML_FLAG_ADD
All added XML nodes in "target" and all its descendants

XML_FLAG_CHANGE
All changed XML nodes in both "src" and "target" and all its descendants. Also all ancestors of all added, deleted and changed nodes.

For example, assume the tree (A B) is replaced with (B C), then the two trees are marked with the following flags:

```
         src            target
          o CHANGE        o CHANGE
          |               |
```

(continues on next page)

(continued from previous page)

```
           o CHANGE        o CHANGE
          /|               |\
DELETE   A B               B C ADD
```

You can use functions, such as `xpath_vec_flag()` to query for changed nodes:

```
if (xpath_vec_flag(xcur, nsc, "//symbol/foo", XML_FLAG_ADD, &vec, &veclen) < 0)
   err;
for (i=0; i<veclen; i++){
   xn = vec[i];
      ...
}
```

7.7 7.7 Privileges

The backend process itself does not really require any specific access, but it may be an important topic for an application using clixon when the plugins are designed. A plugin may need to access privileged system resources (such as configure files).

The backend itself is usually started as root: *sudo clixon_backend -s init*, which means that the plugins also run as root (being part of the same process).

The backend can also be started as a non-root user. However, you may need to set some config options to allow user write access, for example as follows(there may be others):

```
<CLICON_SOCK>/tmp/example.sock</CLICON_SOCK>
<CLICON_BACKEND_PIDFILE>/tmp/mytest/example.pid</CLICON_BACKEND_PIDFILE>
<CLICON_XMLDB_DIR>/tmp/mytest</CLICON_XMLDB_DIR>
```

7.7.1 7.7.1 Dropping privileges

You may want to start the backend as root and then drop privileges to a non-root user which is a common technique to limit exposure of exploits.

This can be done either by command line-options: *sudo clicon_backend -s init -U clicon* or (more generally) using configure options:

```
<CLICON_BACKEND_USER>clicon</CLICON_BACKEND_USER>
<CLICON_BACKEND_PRIVILEGES>drop_perm</CLICON_BACKEND_PRIVILEGES>
```

This will initialize resources as root and then *permanently* drop uid:s to the unprivileged user (*clicon* in the example abobe). It will also change ownership of several files to the user, including datastores and the clicon socket (if the socket is unix domain).

Note that the unprivileged user must exist on the system, see *Install section*.

7.7.2 7.7.2 Drop privileges temporary

If you drop privileges permanently, you need to access all privileged resources initially before the drop. For a plugin designer, this means that you need to access privileges system resources in the *plugin_init* or *plugin_start* callbacks. The transaction callbacks, for example, will be run in unprivileged mode.

An alternative is to drop privileges temporary and then be able to raise privileges when needed:

```
<CLICON_BACKEND_USER>clicon</CLICON_BACKEND_USER>
<CLICON_BACKEND_PRIVILEGES>drop_temp</CLICON_BACKEND_PRIVILEGES>
```

In this mode, a plugin callback (eg commit), can temporarily raise the privileges when accessing system resources, and the lower them when done.

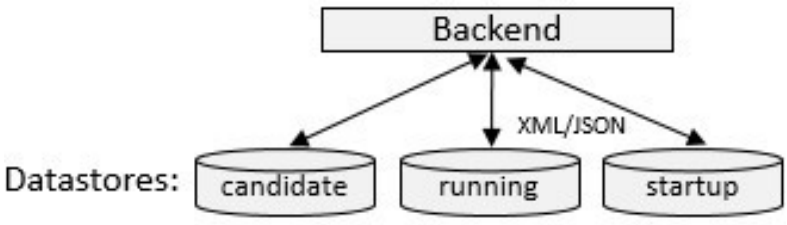
An example C-code for raising privileges in a plugin is as follows:

```
uid_t euid = geteuid();
restore_priv();
... make high privilege stuff...
drop_priv_temp(euid);
```

CHAPTER

EIGHT

8 DATASTORE



Clixon configuration datastores follow the Netconf model (from RFC 6241: NETCONF Configuration Protocol):

Candidate
: A configuration datastore that can be manipulated without impacting the device's current configuration and that can be committed to the running configuration datastore.

Running
: A configuration datastore holding the complete configuration currently active on the device.

Startup
: The configuration datastore holding the configuration loaded by the device when it boots. Only present on devices that separate the startup configuration datastore from the running configuration datastore.

There are also other datastores, Clixon is not limited to the three datastores above. For example:

tmp
: The tmp datastore appears in several cases as an intermediate datastore.

Rollback
: If the confirmed-commit feature is enabled, the rollback datastore holds the running datastore as it existed before the confirm commit. If a cancel or timeout occurs, the rollback datastore is used to revert to.

8.1 8.1 Datastore files

The mandatory *CLICON_XMLDB_DIR* option determines where the datastores are placed. Example:

```
<CLICON_XMLDB_DIR>/usr/local/var/example</CLICON_XMLDB_DIR>
```

The permission of the datastores files is accessible to the user that starts the backend only. Typically this is *root*, but if the backend is started as a non-privileged user, or if privileges are dropped (see *Backend section*) this may be another user, such as in the following example where *clicon* is used:

```
sh> ls -l /usr/local/var/example
-rwx------ 1 clicon clicon   0 sep 15 17:02 candidate_db
```

(continues on next page)

(continued from previous page)

```
-rwx------ 1 clicon clicon   0 sep 15 17:02 running_db
-rwx------ 1 clicon clicon   0 sep 14 18:12 startup_db
```

Note that a user typically does not access the datastores directly, it is possible to read, but write operations should not be done, since the backend daemon may use a datastore cache, see *Datastore caching*.

8.2 8.2 Datastore and file formats

By default, the datastore files use pretty-printed XML, with the top-symbol *config*. The following is an example of a valid datastore:

```
<config>
  <hello xmlns="urn:example:hello">
    <world/>
  </hello>
</config>
```

The format of the datastores can be changed using the following options:

CLICON_XMLDB_FORMAT
: Datastore format. *xml* is the primary alternative. *json* is also available, while *text* and *cli* are available as file formats but not specifically for the datastore.

CLICON_XMLDB_PRETTY
: XMLDB datastore pretty print. The default value is *true*, which inserts spaces and line-feeds making the XML/JSON human readable. If false, the XML/JSON is more compact.

Note that the format settings applies to all datastores.

8.2.1 8.2.1 Other formats

While only XML and JSON are currently supported as datastore formats, Clixon also supports *CLI* and *TEXT* formats for printing, and saving and loading files.

The main example contains example code showing how to load and save a config using other formats.

Example of showing a config as XML, JSON, TEXT and CLI:

```
cli> show configuration xml
<table xmlns="urn:example:clixon">
   <parameter>
      <name>a</name>
      <value>17</value>
   </parameter>
   <parameter>
      <name>b</name>
      <value>99</value>
   </parameter>
</table>
cli> show configuration json
{
  "clixon-example:table": {
```

(continues on next page)

(continued from previous page)

```
    "parameter": [
      {
        "name": "a",
        "value": "17"
      },
      {
        "name": "b",
        "value": "99"
      }
    ]
  }
}
cli> show configuration text
clixon-example:table {
    parameter a {
        value 17;
    }
    parameter b {
        value 99;
    }
}
cli> show configuration cli
set table parameter a
set table parameter a value 17
set table parameter b
set table parameter b value 99
```

Save and load a file using TEXT:

```
cli> save foo.txt text
cli> load foo.txt replace text
```

Internal C API

CLI show and save commands uses an internal API for print, save and load of the formats. Such CLI functions include: *cli_show_config*, *cli_pagination*, *load_config_file*, *save_config_file*.

The following internal C API is available for output:

- XML: `clixon_xml2file()` and `clixon_xml2cbuf()` to file and memory respectively.
- JSON: `clixon_json2file()` and `clixon_json2cbuf()`
- CLI: `clixon_cli2file()`
- TEXT: `clixon_txt2file()`

The arguments of these functions are similar with some local variance. For example:

```
int
clixon_xml2file(FILE            *f,
                cxobj           *xn,
                int              level,
                int              pretty,
```

(continues on next page)

(continued from previous page)

```
                clicon_output_cb *fn,
                int              skiptop,
                int              autocliext)
```

where:

- *f* is the output stream (such as *stdout*)
- *xn* is the top-level XML node
- *level* is indentation level to start with, normally *0*
- *pretty* makes the output indented and use newlines
- *fn* is the output function to use. *NULL* means *fprintf*, *cligen_output* is used for scrolling in CLI
- *skiproot* only prints the children by skipping the top-level XML node *xn*
- *autocliext* Set if you want to activate autocli extensions (eg *hide* extensions)

8.3 8.3 Module library support

Clixon can store Yang module-state information according to RFC 8525: YANG library in the datastores. With module state, you know which Yang version the XML belongs to, which is useful when upgrading, see *upgrade*.

To enable yang module-state in the datastores add the following entry in the Clixon configuration:

```
<CLICON_YANG_LIBRARY>true</CLICON_YANG_LIBRARY> # (default true)
<CLICON_XMLDB_MODSTATE>true</CLICON_XMLDB_MODSTATE>
```

If the datastore does not contain module-state, general-purpose upgrade is the only upgrade mechanism available.

A backend with *CLICON_XMLDB_MODSTATE* disabled will silently ignore module state.

Example of a (simplified) datastore with Yang module-state:

```
<config>
  <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <content-id>42</content-id>
    <module-set>
      <name>default</name>
      <module>
        <name>A</name>
        <revision>2019-01-01</revision>
        <namespace>urn:example:a</namespace>
      </module>
    </module-set>
  </yang-library>
  <a1 xmlns="urn:example:a">some text</a1>
</config>
```

Note that the module-state is not available to the user, the backend datastore handler strips the module-state info. It is only shown in the datastore itself.

8.4 8.4 Datastore caching

Clixon datastore cache behaviour is controlled by the *CLICON_DATASTORE_CACHE* and can have the following values:

nocache
: No cache, always read and write directly with datastore file.

cache
: Use in-memory write-through cache. Make copies of the XML when accessing internally by callbacks and plugins. This is the default.

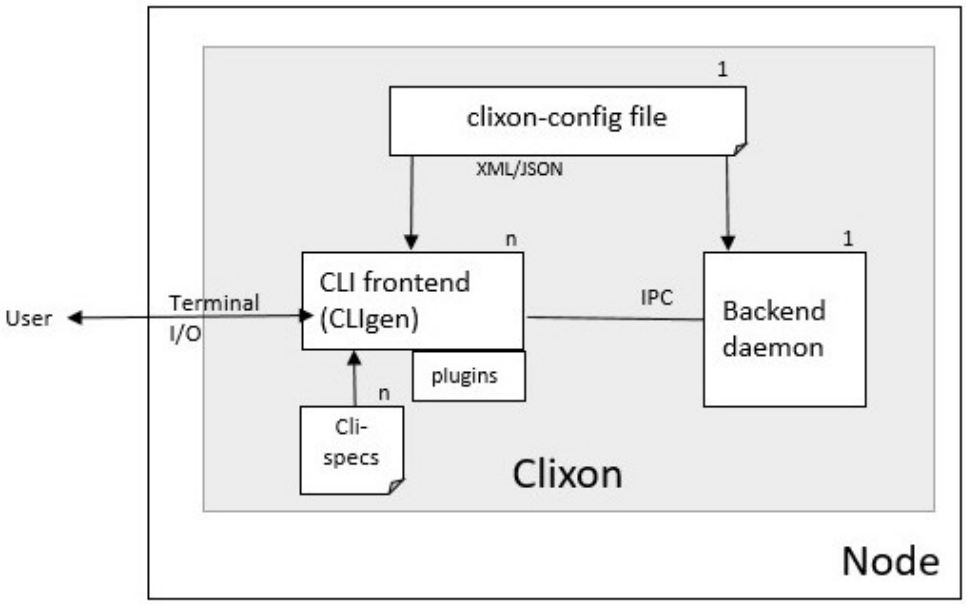
cache-zerocopy
: Use in-memory write-through cache and do not copy when doing callbacks. This is the fastest but opens up for callbacks changing the cache. That is, plugin callbacks may not edit the XML in any way.

Note: Netconf locks are not supported for nocache mode

CHAPTER

NINE

9 CLI



9.1 9.1 Overview

The Clixon CLI provides an interactive command-line interface to a user. Each usage instantiates a new process which communicates via NETCONF with the backend daemon over an IPC socket.

The Clixon CLI uses CLIgen, an interactive interpreter of commands. Syntax is given as *cli-specifications* which specify callbacks defined in plugins.

For details on CLIgen syntax and behavior, please consult the CLIgen tutorial.

Clixon comes with a generated CLI, the *autocli*, where all configuration-related syntax is generated from YANG.

You can also create a completely manually-made CLI.

The CLI depends on the following:

- *Clixon-config*: The Clixon config-file contains initial CLI configuration, such as where to find cli-specs, plugins and autocli configuration.
- *Cli-specs*: CLI specification files written in CLIgen syntax.
- *Plugins*: Dynamic loadable plugin files loaded at startup. Callbacks from cli-spec files are resolved and need to exist as symbols either in the Clixon libs or in the plugin file.

The following example from the main example. First, a cli-spec file containing two commands:

```
set("Set configuration symbol") @datamodel, cli_auto_set();
show("Show a particular state of the system") configuration("Show configuration"), cli_
↪show_config("candidate", "text", "/");
example("Callback example") <var:int32>("any number"), mycallback("myarg");
```

In the CLI, these generate CLI commands such as:

```
set interfaces interface eth9
show config
example 23
```

The effect of typing the commands above is calling callbacks, either library functions in Clixon libs(`cli_show_config()`), or application-defined in a plugin(`mycallback()`)

In this way, a designer writes cli command specifications which invokes C-callbacks. If there are no appropriate callbacks the designer must write a new callback function.

9.1.1 9.1.1 Example usage

The following example shows an auto-cli session from the main example how to add an interface in candidate, validate and commit it to running, then look at it as xml and cli and finally delete it:

```
clixon_cli -f /usr/local/etc/example.xml
user@host> set interfaces interface eth9 ?
  description               enabled                   ipv4
  ipv6                      link-up-down-trap-enable  type
user@host> set interfaces interface eth9 type ex:eth
user@host> validate
user@host> commit
user@host> show configuration xml
<interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
  <interface>
    <name>eth9</name>
    <type>ex:eth</type>
    <enabled>true</enabled>
  </interface>
</interfaces>
user@host> show configuration cli
set interfaces interface eth9
set interfaces interface eth9 type ex:eth
set interfaces interface eth9 enabled true
user@host> delete interfaces interface eth9
```

9.1.2 9.1.2 Command-line options

The *clixon_cli* client has the following command-line options:

-h	Help
-V	Show version and exit
-D <level>	Debug level
-f <file>	Clixon config file
-E <dir>	Extra configuration directory
-l <option>	Log on (s)yslog, std(e)rr, std(o)ut, (n)one or (f)ile. Stderr is default.
-C <format>	Dump configuration options on stdout after loading. Format is one of xml|json|text
-F <file>	Read commands from file (default stdin)
-1	Run once, do not enter interactive mode
-a <family>	Internal IPC backend socket family: UNIX|IPv4|IPv6
-u <path|addr>	Internal IPC socket domain path or IP addr (see -a)
-d <dir>	Specify cli plugin directory
-m <mode>	Specify plugin syntax mode
-q	Quiet mode, do not print greetings or prompt, terminate on ctrl-C
-p <dir>	Add Yang directory path (see CLICON_YANG_DIR)
-G	Print auo-cli CLI syntax generated from YANG
-L	Debug print dynamic CLI syntax including completions and expansions
-y <file>	Load yang spec file (override yang main modul)e
-c <file>	Specify cli spec file
-U <user>	Over-ride unix user with a pseudo user for NACM.
-o <option=value>	Give configuration option overriding config file (see clixon-config.yang)

Inline CLI commands

CLI commands can be given directly after the options. These are executed directly:

```
clixon_cli -f example.xml show config
```

One can also add extra application-dependent plugin options after – which can be read with *clicon_argv_get()*:

```
clixon_cli -f example.xml show config -- -x extra-option
```

9.2 9.2 Configure options

The following config options are related to clispec and plugin files (clixon config options), ie they are set in the XML Clixon config file:

CLICON_CLI_DIR
Directory containing frontend cli loadable plugins. Load all *.so* plugins in this directory as CLI object plugins.

CLICON_CLISPEC_DIR
Directory containing frontend cligen spec files. Load all *.cli* files in this directory as CLI specification files.

CLICON_CLISPEC_FILE
Specific frontend cligen spec file as alternative or complement to *CLICON_CLISPEC_DIR*. Also available as *-c* in clixon_cli.

9.2.1 9.2.1 Terminal I/O

Clixon CLI have the following configuration options related to terminal I/O:

CLICON_CLI_LINESCROLLING
Set to *0* if you want CLI to wrap to next line. Set to *1* if you want CLI to scroll sideways when approaching right margin (default).

CLICON_CLI_LINES_DEFAULT Set to number of CLI terminal rows for pagination/scrolling. *0* means unlimited. The number is set statically UNLESS:

- there is no terminal, such as file input, in which case nr lines is *0*
- there is a terminal sufficiently powerful to read the number of lines from ioctl calls.

In other words, this setting is used ONLY on raw terminals such as serial consoles.

CLICON_CLI_TAB_MODE
Set CLI tab mode. See detailed info in YANG source

9.2.2 9.2.2 History

Clixon CLI supports persistent command history. There are two CLI history related configuration options:

CLICON_CLI_HIST_FILE
The file containing the history, default value is: *~/.clixon_cli_history*

CLICON_CLI_HIST_SIZE
Max number of history line, default value is 300.

The design is similar to bash history but is simpler in some respects:

- The CLI loads/saves its complete history to a file on entry and exit, respectively
- The size (number of lines) of the file is the same as the history in memory
- Only the latest session dumping its history will survive (bash merges multiple session history).

Further, tilde-expansion is supported and if history files are not found or lack appropriate access will not cause an exit but are logged at debug level

9.2.3 9.2.3 Help strings

Help strings are specified using the following example syntax: `("help string")`. help strings are shown at queries, eg "?":

```
user@host> show <?>
   all       Show all
   routing   Show routing
   files     Show files
```

For long or multi-line help strings the following configure options exists:

CLICON_CLI_HELPSTRING_TRUNCATE
: Set to 0 to wrap long help strings to the next line. (default) Set to 1 to truncate long help strings at the right margin

CLICON_CLI_HELPSTRING_LINES
: Set to 0 to have no limit on the number of help string lines per command Set to <n> to limit the the number of help string lines

Long and multi-line help strings may especially be needed in the auto-cli, see *autocli*.

9.2.4 9.2.4 Modes

The CLI can have different *modes* which is controlled by a config option and some internal clispec variables. The config options are:

CLICON_CLI_MODE
: Startup CLI mode. This should match a `CLICON_MODE` variable setting in one of the clispec files. Default is "base".

CLICON_CLI_VARONLY
: Do not include keys in cvec in cli vars callbacks

Inside the clispec files `CLICON_MODE` is used to specify to which modes the syntax in a specific file defines. For example, if you have major modes *configure* and *operation* you can have a file with commands for only that mode, or files with commands in both, (or in all).

First, lets add a single command in the configure mode:

```
CLICON_MODE="configure";
show configure;
```

Then add syntax to both modes:

```
CLICON_MODE="operation:configure";
show("Show") files("Show files");
```

Finally, add a command to all modes:

```
CLICON_MODE="*";
show("Show") all("Show all");
```

Note that CLI command trees are merged so that show commands in other files are shown together. Thus, for example, using the clispecs above the two modes are the three commands in total for the *configure* mode:

```
> clixon_cli -m configure
user@host> show <TAB>
  all     routing      files
```

but only two commands in the *operation* mode:

```
> clixon_cli -m operation
user@host> show <TAB>
  all      files
```

9.2.5 9.2.5 Cli-spec variables

A CLI specification file (note not clixon config file) typically starts with the following variables:

CLICON_MODE
: A colon-separated list of CLIgen *modes*. The CLI spec in the file are added to *all* modes specified in the list. You can also use wildcards * and '*?*'.

CLICON_PROMPT
: A string describing the CLI prompt using a very simple format with: `%H` (host) , `%U` (user) , `%T` (tty), `%W` (last element of working path), `%w` (full working path).

CLICON_PLUGIN
: The name of the object file containing callbacks in this file.

CLICON_PIPETREE
: Name of a pipe output tree as described in

9.3 9.3 CLI callbacks

CLI callback functions are "library" functions that an application may call from a clispec. A user is expected to create new application-specific callbacks.

As an example, consider the following clispec:

```
example("Callback example") <var:int32>("any number"), mycallback("myarg");
```

containing a keyword (*example*) and a variable (*var*) and *mycallback* is a cli callback with argument: (*myarg*).

In C, the callback has the following signature:

```
int mycallback(clixon_handle h, cvec *cvv, cvec *argv);
```

Suppose a user enters the following command in the CLI:

```
user@host> example 23
```

The callback is called with the following parameters:

```
cvv:
   0: example 23
   1: 23
argv:
   0: "myarg"
```

which means that *cvv* contains dynamic values set by the user, and *argv* contains static values set by the clispec designer.

9.4 9.4 Show commands

Clixon includes show commands for showing datastore and state content.An application may use these functions as basis for more specialized show functions. Some show functions are:

- `cli_show_config()` - Multi-purpose show function for manual CLI show commands
- `cli_show_auto()` - Used in conjunction with the autocli with expansion trees
- `cli_show_auto_mode()` - Used in conjunction with the autocli with edit-modes
- `cli_pagination()` - Show paginated data of a large list

The CLI show functions are utility functions in the sense that they are not part of the core functionality and a user or product may want to specialize them.

Note: CLI library functions are subject to change in new releases

9.4.1 9.4.1 cli_show_config

The `cli_show_config` is a basic function to display datastore and state data. A typical use in a cli spec is as follows:

```
show("Show configuration"), cli_show_config("candidate", "text");
```

Using this command in the CLI could provicde the following output:

```
cli> show
<table xmlns="urn:example:clixon">
   <parameter>
      <name>a</name>
      <value>x</value>
   </parameter>
</table>
```

The callback has the following parameters, only the first is mandatory:

- *dbname* : Name of datastore to show, such as "running", "candidate" or "startup"
- *format* : Show format, one of *text*, *xml*, *json*, *cli*, or *netconf* (see *datastore formats*)
- *xpath* : Static xpath (only present in this API function)
- *namespace* : Default namespace for xpath (only present in this API function)
- *pretty* : If *true*, make output pretty-printed
- *state* : If *true*, include non-config data in output
- *default* : Optional default retrieval mode: one of *report-all*, *trim*, *explicit*, *report-all-tagged*. See also extended values below
- *prepend* : Optional prefix to prepend before cli syntax output, only valid for CLI format.
- *fromroot* : If *false* show from xpath node, if *true* show from root

9.4.2 9.4.2 cli_show_auto

The `cli_show_auto()` callback is used together with the autocli to show sub-parts of a configured tree using expansion. A typical definition is as follows:

```
show("Show expand") @datamodelshow, cli_show_auto("candidate", "xml");
```

That is, it must always be used together with a tree-reference as described in Section *autocli*.

An example CLI usage is:

```
cli> show table parameter a
<parameter>
   <name>a</name>
   <value>x</value>
</parameter>
```

The arguments are similar to *cli_show_config* with the difference that the *xpath* is implicitly defined by the current position of the tree reference:

```
* `dbname` : Name of datastore to show, such as "running", "candidate" or "startup"
* `format` : Show format, one of `text`, `xml`, `json`, `cli`, or `netconf` (see␣
↪:ref:`datastore formats <clixon_datastore>`)
* `pretty` : If `true`, make output pretty-printed
* `state`  : If `true`, include non-config data in output
* `default` : Optional default retrieval mode: one of `report-all`, `trim`, `explicit`,␣
↪`report-all-tagged`. See also extended values below
* `prepend`  : Opional prefix to print before cli syntax output, only valid for CLI␣
↪format.
```

9.4.3 9.4.3 cli_show_auto_mode

The `cli_show_auto_mode()` callback also used together with the autocli but instead of exapansion uses the edit-modes (see Section *edit modes*). A typical definition is:

```
show, cli_show_auto_mode("candidate");
```

An example usage using edit-modes is:

```
 cli> edit table
 cli> show
<parameter>
   <name>a</name>
   <value>x</value>
</parameter>
```

Same parameters as `cli_show_auto`

9.4.4 9.4.4 Common show parameters

with-default parameter

All show commands have an optional *with-default* retrieval mode: one of *report-all*, *trim*, *explicit*, *report-all-tagged*. There are also extra propriatary modes of *default* serving as examples:

- *NULL*, default with-default value, usually *report-all*
- *report-all-tagged-default*, which gets the config as *report-all-tagged* but strips the tags/attributes (same as *report-all*).
- *report-all-tagged-strip*, which also gets the config as *report-all-tagged* but strips the nodes associated with the default tags (same as *trim*).

pretty parameter

All show commands have a pretty-print parameter. If *true* the putput is pretty-printed. Indentation level is controlled by the `PRETTYPRINT_INDENT` compile-time option

9.5 9.5 Output pipes

Output pipes resemble UNIX shell pipes and are useful to filter or modify CLI output. Example:

```
cli> print all | grep parameter
  <parameter>5</parameter>
  <parameter>x</parameter>
cli> show config | showas json
  {
    "table": {
      "parameter": [
          ...
cli>
```

Output pipe functions are declared using a special variant of a CLI tree with a name starting with a *vertical bar*. Example:

```
CLICON_MODE="|mypipe";
\| {
   grep <arg:rest>, pipe_grep_fn("-e", "arg");
   showas json, pipe_json_fn();
}
```

where `pipe_grep_fn` and `pipe_json_fn` are special callbacks that use stdio to modify output.

Such a pipe tree can be referenced with either an explicit reference, or an implicit rule.

Only a single level of pipes is possibly in this release. For example, `a|b|c` is *not* possible.

Note: Only one level of pipes is supported

9.5.1 9.5.1 Explicit reference

An explicit reference is for single commands. For example, adding a pipe to the print commands:

```
print, print_cb("all");{
   @|mypipe, print_cb("all");
   all @|mypipe, print_cb("all");
   detail;
}
```

where a pipe tree is added as a tree reference, appending pipe functions to the regular `print_cb` callback. Note that the following commands are possible in this example:

```
print
print | count
print all | count
print detail
```

9.5.2 9.5.2 Implicit rule

An implicit rule adds pipes to *all* commands in a cli mode. An example of an implicit rule is as follows:

```
CLICON_PIPETREE="|mypipe";
print, print_cb("all");{
   all, print_cb("all");
   detail, print_cb("detail);
}
```

where the pipe tree is added implicitly to all commands in that file, and possibly on other files with the same mode.

Pipe trees also work for sub-trees, ie a subtree referenced by the top-level tree may also use output pipes.

9.5.3 9.5.3 Combinations

It is possible to combine an implicit (default) rule with an explict rule as follows:

```
CLICON_MODE="|commonpipe";
print, print_cb("all");{
   @|mypipe, print_cb("all");
   all @|mypipe, print_cb("all");
   detail;
}
```

In this example, *print* and *print all* use the *|mypipe* menu, while *print detail* uses the *|common* menu

9.5.4 9.5.4 Inheriting

Sub-trees inherit pipe commands from the top-level according to the following rules:

1. Top-level implicit rules are inherited to all sub-trees, unless
2. Explicit rules are present at the tree-reference
3. No pipe commands are allowed in a pipe-command (only single level allowed)

Rules 1 and 2 are illustrated as follows:

```
CLICON_MODE="|commonpipe";
aaa {
   @datamodel, cli_show();
   @|mypipe, cli_show();
}
bbb {
   @datamodel, cli_show();
}
```

Pipe commands in the *datamodel* tree are |*mypipe* if preceeded by *aaa*, but |*commonpipe* if preceeded by *bbb*

9.5.5 9.5.5 Pipe functions

Clixon contains several example pipe functions primarily for testing, users of Clixon should review these and consider implementing their own.

9.6 9.6 Autocli

The Clixon CLI contains parts that are *generated* from a YANG specification. This *autocli* is generated from YANG into CLI specifications, parsed and merged into the top-level Clixon CLI.

The autocli is configured using three basic mechanisms:

1. *Config file* : Modify behavior of the generated tree
2. *Tree expansion*: How the generated cli is merged into the overall CLI
3. *YANG Extensions*: Modify CLI behavior via YANG

Each mechanism is described in sub-sections below, but first an overview of autocli usage.

9.6.1 9.6.1 Overview

Consider a (simplified) YANG specification, such as:

```
module example {
  container table {
    list parameter{
      key name;
      leaf name{
        type string;
      }
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

An example of a generated syntax is as follows (again simplified):

```
table; {
   parameter <name:string>;
}
```

The auto-cli syntax is loaded using a *sub-tree operator* such as `@datamodel` into the Clixon CLI as follows:

```
CLICON_PROMPT="%U@%H %W> ";
set @datamodel, cli_auto_set();
merge @datamodel, cli_auto_merge();
delete @datamodel, cli_auto_del();
show config, cli_auto_show("datamodel", "candidate", "text", true, false);{
   @datamodel, cli_show_auto("candidate", "text");
}
```

For example, the *set* part is expanded using the CLIgen tree-operator to something like:

```
set table, cli_auto_set(); {
      parameter <name:string>, cli_auto_set();
}
```

An example run of the above example is as follows:

```
> clixon_cli
user@host /> set table ?
  <cr>
  parameter
user@host /> set table parameter 23
user@host /> show config
table {
   parameter {
      name 23;
   }
}
user@host />
```

where the generated autocli extends the Clixon CLI with YANG-derived configuration statements.

9.6.2 9.6.2 Config file

The clixon config file has a <autocli> sub-clause for global autocli configurations. A typical CLI configuration with default autocli settings is as follows:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_CONFIGFILE>/usr/local/etc/example.xml</CLICON_CONFIGFILE>
  ...
  <autocli>
```

(continues on next page)

(continued from previous page)

```
    <module-default>true</module-default>
    <list-keyword-default>kw-nokey</list-keyword-default>
    <treeref-state-default>false</treeref-state-default>
    <edit-mode-default>list container</edit-mode-default>
    <completion-default>true</completion-default>
  </autocli>
</clixon-config>
```

The autocli configuration consists of a set of default *options*, followed by a set of *rules*. For more info see the `clixon-autocli.yang` specification.

Options

The following options set default values to the auto-cli, some of these may be further refined by successive rules.

module-default
How to generate the autocli from modules:

- If *true*, all modules with a top-level datanode are generated, ie they get a top-level entry in the `@basemodel` tree. This is default
- If *false*, you need to explicitly enable modules for autocli generation using *module enable rules*.

list-keyword-default
How to generate the autocli from YANG lists. There are several variants defined. To understand the different variants, consider a simple YANG LIST defintion as follows:

```
list a {
   key x;
   leaf x;
   leaf y;
}
```

The different variants with the resulting autocli are as follows:

- *kw-none* : No extra keywords, only variables: `a <x> <y>`
- *kw-nokey* : Keywords on non-key variables: `a <x> y <y>`. This is default.
- *kw-all* : Keywords on all variables: `a x <x> y <y>`

treeref-state-default
If generate autocli from YANG *state* data. The motivation for this option is that many specs have very large state parts. In particular, some openconfig YANG specifications have ca 10 times larger state than config parts.

- If *true*, generate CLI from YANG state/non-config statements, not only from config data.
- If *false* do not generate autocli commands from YANG state data. This is default.

edit-mode-default
Open automatic edit-modes for some YANG keywords and do not allow others. A CLI edit mode opens a carriage-return option and changes the context to be in that local context. For example:

```
user@host> interfaces interface e0<cr>
eth0>
```

Default is to generate edit-modes for all YANG containers and lists. For more info see *edit modes*

completion-default
: Generate code for CLI completion of existing db symbols. That is, check existing configure database for completion options. This is normally always enabled.

grouping-treeref
: Controls the behaviour when generating CLISPEC of YANG 'uses' statements into the corresponding 'grouping' definition. If 'true', use indirect tree reference '@treeref' to reference the grouping definition. This may reduces memory footprint of the CLI.

Rules

To complement options, a set of rules to further define the autocli can be defined. Common rule fields are:

name
: Arbitrary name assigned for the rule, must be unique.

operation
: Rule operation, There are currently two operations defined: *module enable* and command *compress*.

module-name
: Name of the module associated with this rule. Wildchars '*' and '?' can be used (glob pattern). Revision and yang suffix are omitted. Example: `openconfig-*`

Module enable rules

Module enable rules are used in combination with `module-default=false` to enable CLI generation for a limited set of YANG modules.

For example, assume you want to enable modules *example1*, *example2* and no others:

```
<autocli>
   <module-default>false</module-default>
   <rule>
      <name>include example</name>
      <operation>enable</operation>
      <module-name>example*</module-name>
  </rule>
</autocli>
```

If the option `module-default` is `true`, module enable rules have no effect since all modules are already enabled.

Compress rules

Compress rules are used to skip CLI commands, making the complete command name shorter.

For example, assume YANG definition:

```
container interfaces {
   list interface {
      ...
   }
}
```

Instead of typing `interfaces interface e0` you would want to type only `interface e0`. The following rule matches all YANG containers with lists as its only child, and removes the keyword `interfaces`:

```
<rule>
   <name>compress</name>
   <operation>compress</operation>
   <yang-keyword>container</yang-keyword>
   <yang-keyword-child>list</yang-keyword-child>
</rule>
```

Note that this matches the openconfig compress rule: The surrounding 'container' entities are removed from 'list' nodes

A second openconfig compress rule is The 'config' and 'state' containers are "compressed" out of the schema. as examplified here (for 'config' only):

```
<rule>
   <name>openconfig compress</name>
   <operation>compress</operation>
   <yang-keyword>container</yang-keyword>
   <schema-nodeid>config</schema-nodeid>
   <module-name>openconfig*</module-name>
</rule>
```

Specific fields for compress are:

yang-keyword
: If present identifes a YANG keyword which the rule applies to. Example: `container`

schema-nodeid
: A single <id> identifying a YANG schema-node identifier as defined in RFC 7950 Sec 6.5. Example: `config`

yang-keyword-child
: The YANG statement has a single child, and the yang type of the child is the value of this option. Example: : `container`

extension
: The extension is set either in the node itself, or in the module the node belongs to. Extension prefix must be set. Example: `oc-ext:openconfig-version`

9.6.3 9.6.3 Tree expansion

In the example above, the tree-reference `@datamodel` is used to merge the YANG-generated cli-spec into the overall cli-spec. There are several variants of how the generated tree is expanded with slight differences in which symbols are shown, how completion works, etc.

They are all derivates of the basic `@basemodel` tree. The following tree variants are defined:

- `@basemodel` - The most basic tree including everything
- `@datamodel` - The most common tree for configuration with state
- `@datamodelshow` - A tree made for showing configuration syntax
- `@datamodelmode` - A tree for editing modes
- `@datamodelstate` - A tree for showing state as well as configuration

Note to use `@datamodelstate` config option `treeref-state-default` must be set.

9.6.4 9.6.4 YANG Extensions

A third method to define the autocli is using *YANG extensions*, where a YANG specification is annotated with extension.

Clixon provides a dedicated YANG extension for the autocli for this purpose: `clixon-lib:autocli`.

The following example shows the main example usage of the "hide" extension of the "hidden" leaf:

```
import clixon-autocli{
   prefix autocli;
}
container table{
   list parameter{
      ...
      leaf hidden{
         type string;
         autocli:hide;
      }
   }
}
```

The CLI `hidden` command is not shown but the command still exists:

```
cli /> set table parameter a ?
value
<cr>
cli /> set table parameter a hidden 99
cli /> show configuration
table {
  parameter {
      name a;
      hidden 99;
  }
}
```

The following autocli extensions are defined:

hide
: Do not show the command in eg auto-completion. This was primarily intended for operational commands such as `start shell` but is this context used for hiding commands generated from the associated YANG node.

skip
: Skip the command altogether.

hide-show
: Do not show the config in show configuration commands. However, retreiving a config via NETCONF or examining the datastore directly shows the hidden configure commands.

strict-expand
: Only show exactly the expanded options of a variable. It shuld not be possible to add a *new* value that is not in the expanded list.o

alias
: Replace the command with another value, only implemented for YANG leaves.

9.6.5 9.6.5 Edit modes

The autocli supports *automatic edit modes* where by entering a <cr>, you enter an edit mode. An edit mode is created for every YANG container or list.

For example, the example YANG previously given and the following cli-spec:

```
edit @datamodelmode, cli_auto_edit("basemodel");
up, cli_auto_up("basemodel");
top, cli_auto_top("basemodel");
set @datamodel, cli_auto_set();
```

Then an example session for illustration is as follows, where first a small config is created, then a list instance mode is entered(`parameter a`), a value changed, and a container mode (`table`):

```
user@host /> set table parameter a value 42
user@host /> set table parameter b value 77
user@host /> edit table parameter a
user@host parameter=a/>
user@host parameter=a/> show configuration
  name a;
  value 42;
user@host parameter=a/> set value 99
user@host parameter=a/> up
user@host table> show configuration
parameter {
    name a;
    value 99;
}
parameter {
    name b;
    value 77;
}
user@host table> top
user@host />
```

9.7 9.7 Advanced

This section describes some advanced options in the Clixon CLI not described elsewhere.

9.7.1 9.7.1 Backend socket

By default, the CLI uses a UNIX socket as an IPC to communicate with the backend. It is possible to use an IP socket but with a restricted functionality, see *backend section*.

Start session

The session creation is "lazy" in the sense that a NETCONF session is only established when needed. After the session has been established, it is maintained (cached) by the CLI client to keep track of candidate edits and locks, as described in 7.5 of RFC 6241.

If there is no backend running at the time of session establishment, a warning is printed:

```
cli /> show config
Mar 18 11:53:43: clicon_rpc_connect_unix: 541: Protocol error: /usr/local/var/example/
↪example.sock: config daemon not running?: No such file or directory
Protocol error: /usr/local/var/example/example.sock: config daemon not running?: No such␣
↪file or directory
cli />
```

If at a later time, the backend is started, the session is established normally

Close session

After a session is established and the *backend* exits, crashes or restarts, any state associated with the session will be lost, including:

- explicit locks
- edits in candidate-db

If the backend exits during an existing session, it will close with the same error message as above:

```
cli /> show config
Mar 18 11:53:43: clicon_rpc_connect_unix: 541: Protocol error: /usr/local/var/example/
↪example.sock: config daemon not running?: No such file or directory
Protocol error: /usr/local/var/example/example.sock: config daemon not running?: No such␣
↪file or directory
cli />
```

If the backend restarts, a new session is created with a warning:

```
cli /> show configuration
Mar 18 11:57:55: The backend was probably restarted and the client has reconnected to␣
↪the backend. Any locks or candidate edits are lost.
cli />
```

Alternative

It is possible to change the default behavior by undefining the compile-option: *#undef PROTO_RESTART_RECONNECT*. If so, the CLI is exited when the existing session is closed in anyway:

```
cli /> show configuration
Mar 18 12:02:57: clicon_rpc_msg: 210: Protocol error: Unexpected close of CLICON_SOCK.␣
↪Clixon backend daemon may have crashed.: Cannot send after transport endpoint shutdown
Protocol error: Unexpected close of CLICON_SOCK. Clixon backend daemon may have crashed.
↪: Cannot send after transport endpoint shutdown
bash#
```

9.7.2 9.7.2 Sub-tree operator

Sub-trees are defined using the tree operator @. Every mode gets assigned a tree which can be referenced as *@name*. This tree can be either on the top-level or as a sub-tree. For example, create a specific sub-tree that is used as sub-trees in other modes:

```
CLICON_MODE="subtree";
subcommand{
  a, a();
  b, b();
}
```

then access that subtree from other modes:

```
CLICON_MODE="configure";
main @subtree;
other @subtree,c();
```

The configure mode will now use the same subtree in two different commands. Additionally, in the *other* command, the callbacks are overwritten by *c*. That is, if *other a*, or *other b* is called, callback function *c* is invoked.

9.7.3 9.7.3 Translators

CLIgen supports wrapper functions that can take the output of a callback and transform it to something else.

The CLI can perform variable translation. This is useful if you want to process the input, such as hashing, encrypting or in other way translate the input.

The following example is based on the main Clixon example and is included in the regression tests. In the following CLI specification, a "translate" command sets a modifed value to the "table/parameter=translate/value":

```
translate <value:string translate:cli_incstr()>, cli_set("/clixon-example:table/
↪parameter=translate/value");
```

If you run this example using the *cli_incstr()* function which increments the characters in the input, you get this result:

```
user@host> translate HAL
user@host> show configuration
table {
   parameter {
      name translate;
      value IBM;
   }
}
```

The example is very simple and based on strings, but can be used also for other types and more advanced functions.

9.7.4 9.7.4 Autocli tree labels

The autocli trees described in *tree expansion* are implemented using filtering of CLIgen labels. While `@basemodel` includes all labels, the other trees have removed some labels.

For most uses, the pre-defined trees above are enough, using explicit label filtering is more powerful.

The currently defined labels are:

- `act-list` : Terminal entries of YANG LIST nodes.
- `act-container` : Terminal entries of YANG CONTAINER nodes.
- `ac-leaf` : Leaf/leaf-list nodes
- `act-prekey` : Terminal entries of LIST leaf keys, except the last keys in multi-key cases.
- `act-lastkey` : Terminal entries of LIST leaf keys, except the last keys in multi-key cases.
- `act-leafconst` : Terminal entries of non-empty non-key YANG LEAF/LEAF_LISTs command nodes.
- `act-leafvar` : Terminal entries of non-key YANG LEAF/LEAF_LISTs variable nodes.
- `ac-state` : Nodes which have YANG `config false` as child
- `ac-config` : Nodes nodes which do not have any state nodes as siblings

Labels with prefix `act_` are *terminal* labels in the sense that they mark a terminal command, ie the node itself; while labels with `ac_` represent the non-terminal, ie the whole sub-tree.

As an example, the `@datamodel` tree is `basemodel` with labels removed as follows:

```
@basemodel, @remove:act-prekey, @remove:act-list, @remove:act-leaf, @remove:ac-state;
```

which is an alternative way of specifying the datamodel tree.

9.7.5 9.7.5 Extensions to CLIgen

Clixon adds some features and structure to CLIgen which include:

- A plugin framework for both textual CLI specifications(.cli) and object files (.so)
- Object files contains compiled C functions referenced by callbacks in the CLI specification. For example, in the cli spec command: *a,fn()*, *fn* must exist in the object file as a C function.
- The CLIgen *treename* syntax does not work.
- A CLI specification file is enhanced with the CLIgen variables *CLICON_MODE*, *CLICON_PROMPT*, *CLICON_PLUGIN* and *CLICON_PIPETREE*.
- Clixon generates a command syntax from the Yang specification that can be referenced as *@datamodel*. This is useful if you do not want to hand-craft CLI syntax for configuration syntax.

Example of *@datamodel* syntax:

```
set    @datamodel, cli_set();
merge  @datamodel, cli_merge();
create @datamodel, cli_create();
show   @datamodel, cli_show_auto("running", "xml");
```

The commands (eg *cli_set*) will be called with the first argument an api-path to the referenced object.

9.7.6 9.7.6 Running CLI scripts

The CLI can run scripts using either the `-1` option for single commands:

```
clixon_cli -1 show version
4.8.0.PRE
```

Or using the `-F <file>` command-line option to redirect input from file

clixon_cli -F file

Or using "shebang":

```
#!/usr/local/bin/clixon_cli -F
show version
quit
```

Two caveats regarding "shebang":

1. The clixon config file is */usr/local/etc/clixon.xml*
2. The mode is *CLICON_CLI_MODE*

You may mod this by using soft links or creating a new executable to use use in the "shebang" with other default values.

9.7.7 9.7.7 How to deal with large specs

CLIgen is designed to handle large specifications in runtime, but it may be difficult to handle large specifications from a design perspective.

Here are some techniques and hints on how to reduce the complexity of large CLI specs:

Sub-modes

The *CLICON_MODE* is used to specify in which modes the syntax in a specific file should be added. For example, if you have major modes *configure* and *operation* you can have a file with commands for only that mode, or files with commands in both, (or in all).

First, lets add a basic set in each:

```
CLICON_MODE="configure";
show configure;
```

and

```
CLICON_MODE="operation";
show configure;
```

Note that CLI command trees are *merged* so that show commands in other files are shown together. Thus, for example:

```
CLICON_MODE="operation:files";
show("Show") files("files");
```

will result in both commands in the operation mode:

```
> clixon_cli -m operation
user@host> show <TAB>
  configure      files
```

but

```
> clixon_cli -m configure
user@host> show <TAB>
  configure
```

Sub-trees

You can also use sub-trees and the the tree operator @. Every mode gets assigned a tree which can be referenced as *@name*. This tree can be either on the top-level or as a sub-tree. For example, create a specific sub-tree that is used as sub-trees in other modes:

```
CLICON_MODE="subtree";
subcommand{
  a, a();
  b, b();
}
```

then access that subtree from other modes:

```
CLICON_MODE="configure";
main @subtree;
other @subtree,c();
```

The configure mode will now use the same subtree in two different commands. Additionally, in the *other* command, the callbacks will be overwritten by *c*. That is, if *other a*, or *other b* is called, callback function *c* will be invoked.

C-preprocessor

You can also add the C preprocessor as a first step. You can then define macros, include files, etc. Here is an example of a Makefile using cpp:

```
C_CPP    = clispec_example1.cpp clispec_example2.cpp
C_CLI    = $(C_CPP:.cpp=.cli
CLIS     = $(C_CLI)
all:     $(CLIS)
%.cli : %.cpp
     $(CPP) -P -x assembler-with-cpp $(INCLUDES) -o $@ $<
```

9.7.8 9.7.8 Bits

The Yang bits built-in type as defined in RFC 7950 Sec 9.7 provides a set of bit names. In the CLI, the names should be given in a white-spaced delimited list, such as `"fin syn rst"`.

The RFC defines a "canonical form" where the bits appear ordered by their position in YANG, but Clixon validation accepts them in any order.

Given them in XML and JSON follows thus, eg XML:

```
<flags>fin rst syn</flags>
```

Clixon CLI does not treat individual bits as "first-level objects". Instead it only validates the whole string of bit names. Operations (add/remove) are made atomically on the whole string.

9.7.9 9.7.9 Api-path-fmt

The clixon CLI uses an internal meta-format called `api_path_fmt` which is used to generate api-paths, as described i Section XML.

An api-path-fmt extends an api-path with `%` flag characters (like `printf`) as follows:

- %s: The value of a cligen-variable
- %k: The key of a YANG list

Example, an explicit clispec expansion variable could be:

```
<name:string expand_dbvar("candidate","/interface=%s/%k")>
```

which could expand to `/interface=eth0/mykey` if "eth0" is given as the "name" variable and "mykey" is the YANG interface list key.

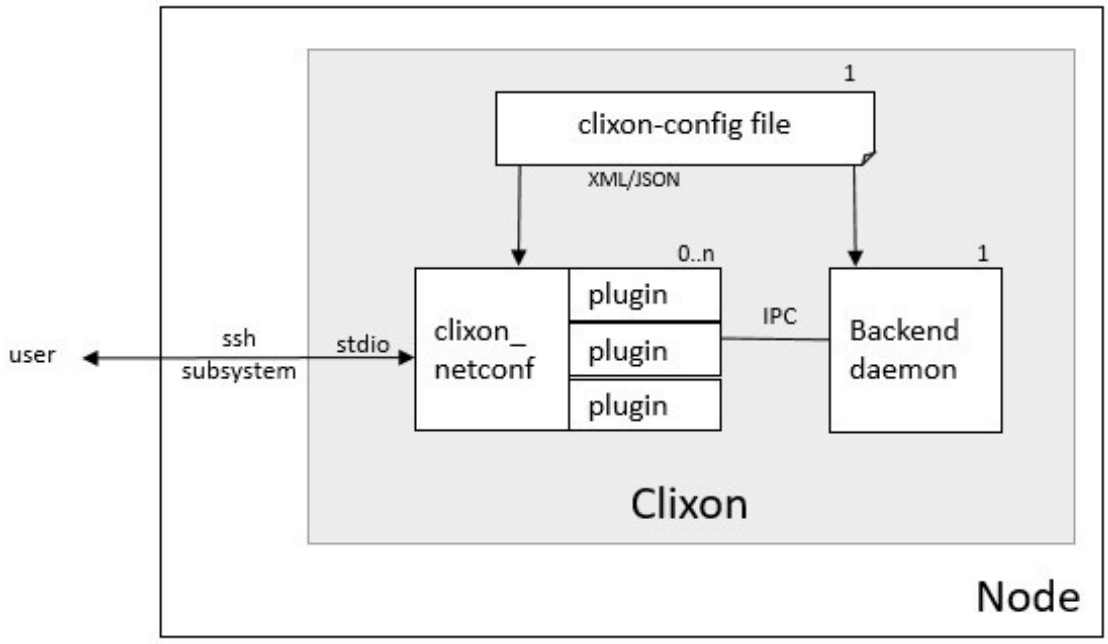
CHAPTER

TEN

10 NETCONF

10.1 10.1 Overview

Netconf is an external client interface (cli and restconf are other external interfaces). Netconf is also used in the internal IPC.



Netconf is defined in RFC 6241 (see *Standards section*) and implemented by the *clixon_netconf* client.

Any number of netconf clients can be created, each creating a new session to the backend. The netconf client communicates to the outside world via *stdio*. Usually one sets up an SSH sub-system to communicate from external nodes.

Note that Netconf supports chunked framing defined in RFC 6241 from Clixon 5.7, but examples may not be updated.

10.1.1 10.1.1 Command-line options

The *clixon_netconf* client has the following command-line options:

-h	Help
-V	Show version and exit
-D <level>	Debug level
-f <file>	Clixon config file
-E <dir>	Extra configuration directory
-l <option>	Log on (s)yslog, std(e)rr, std(o)ut or (f)ile. Syslog is default. If foreground, then syslog and stderr is default.
-C <format>	Dump configuration options on stdout after loading and quit. Format is one of xml|json|text
-q	Quiet mode, server does not send hello message on startup
-0	Set netconf base capability to 0, server does not expect hello, force EOM framing
-1	Set netconf base capability to 1, server does not expect hello, force chunked framing
-a <family>	Internal IPC backend socket family: UNIX|IPv4|IPv6
-u <path|addr>	Internal IPC socket domain path or IP addr (see -a)
-d <dir>	Specify netconf plugin directory
-p <dir>	Add Yang directory path (see CLICON_YANG_DIR)
-y <file>	Load yang spec file (override yang main module)
-U <user>	Over-ride unix user with a pseudo user for NACM.
-t <sec>	Timeout in seconds. Quit after this time.
-e	Do not ignore errors on packet input.
-o <option=value>	Give configuration option overriding config file (see clixon-config.yang)

10.1.2 10.1.2 Configure options

The configuration file options related to NETCONF are the following:

CLICON_NETCONF_DIR
: Location of netconf .so plugins loaded alphabetically

CLICON_NETCONF_HELLO_OPTIONAL
: If true, an RPC can be processed directly with no preceeding hello message. This is not according to the standard RFC 6241 Sec 8.1.

CLICON_NETCONF_MESSAGE_ID_OPTIONAL
: If true, an RPC can be sent without a message-id. This is not according to the standard RFC 6241 Sec 4.1.

10.2 10.2 Starting

The Netconf client (`clixon_netconf`) can be started on the command line using stdin/stdout:

```
> clixon_netconf -qf /usr/local/etc/clixon.conf < my.xml
```

It then reads and parses Netconf commands on stdin, eventually invokes Netconf plugin callbacks, then establishes a connection to the backend and usually sends the Netconf message over IPC to the backend. Some commands (eg hello) are terminated in the client. The reply from the backend is then displayed on stdout.

10.2.1 10.2.1 SSH subsystem

You can expose `clixon_netconf` as an SSH subsystem according to *RFC 6242*. Register the subsystem in `/etc/sshd_config`:

```
Subsystem netconf /usr/local/bin/clixon_netconf
```

and then invoke it from a client using:

```
ssh -s <host> netconf
```

10.3 10.3 NACM

Clixon implements the Network Configuration Access Control Model (NACM / RFC8341). NACM rpc and datanode access validation is supported, not outgoing notifications.

NACM rules apply to all datastores.

10.3.1 10.3.1 Restrictions

Access notification authorization (Sec 3.4.6) is NOT implemented.

Data-node paths, eg `<rule>...<path>ex:table/ex:parameter</path></rule>` instance-identifiers are restricted to canonical namespace identifiers for both XML and JSON encoding. That is, if a symbol (such as `table` above) is a symbol in a module with prefix `ex`, another prefix cannot be used, even though defined with a `xmlns` rule.

10.3.2 10.3.2 Config options

The following configuration options are related to NACM:

CLICON_NACM_MODE
: NACM mode is either: *disabled*, *internal*, or *external*. Default: *disabled*.

CLICON_NACM_FILE
: If NACM mode is external, this file contains the NACM config.

CLICON_NACM_CREDENTIALS
: Verify NACM user credentials with unix socket peer credentials. This means that a NACM user must match a UNIX user accessing `CLIXON_SOCK`. Credentials are either: *none*, *exact* or *except*. Default: *except*.

CLICON_NACM_RECOVERY_USER
RFC8341 defines a 'recovery session' as outside its scope. Clixon defines this user as having special admin rights to exempt from all access control enforcements.

CLICON_NACM_DISABLED_ON_EMPTY
RFC 8341 defines enable-nacm as true by default. Since also write-default is deny by default it leads to that empty configs can not be edited. Default: *false*.

10.3.3 10.3.3 Mode

NACM rules are either internal or external. If external, rules are loaded from a separate file, specified by the option `CLICON_NACM_FILE`.

If the NACM mode is internal, the NACM configuration is a part of the regular candidate/running datastore. NACM rules are read from the *running* datastore, ie they need to be committed.

Since NACM rules are part of the config itself it means that there may be bootstrapping issues. In particular, NACM default is *enabled* with read/exec permit, and write *deny*. Loading an empty config therefore leads to a "deadlock" where no user can edit the datastore.

Work-arounds include restarting the backend with a NACM config in the startup db, or using a *recovery user*.

10.3.4 10.3.4 Access control

NACM is implemented in the Clixon backend at:

- Incoming RPC (module-name/protocol-operation)
- Before modifying the data store (data create/delete/update)
- After retrieving data (data read)

User credentials

Access control relies on a user and groups. When an internal Clixon client communicates with the backend, it piggybacks the name of the user in the request, See *Internal netconf username*:

```
<rpc username="myuser"><get-config><source><running/></source></get-config></rpc>
```

The authentication of the username needs to be done in the client by either SSL certs (such as in *RESTCONF auth callback*) or by SSH (as in NETCONF/CLI over SSH).

The Clixon backend can check credentials of the client if it uses a UNIX socket (not IP socket) for internal communication between clients and backend. In this way, a username claimed by a client can be verified against the UNIX user credentials.

The allowed values of *CLICON_NACM_CREDENTIALS* is:

- *none*: Do not match NACM user to any user credentials. Any user can pose as any other user. Set this for IP sockets, or do not use NACM.
- *exact*: Exact match between NACM user and unix socket peer user.
- *except*: Exact match between NACM user and unix socket peer user, *except* for root and *wwwuser*. This is default.

10.3.5 10.3.5 Recovery user

RFC 8341 defines a NACM emergency recovery session mechanism. Clixon implements a recovery user set by option `CLICON_NACM_RECOVERY_USER`. If a client accesses the backend as that user, all NACM rules will be bypassed. By default there is no such user.

Moreover, this mechanism is controlled by *user credentials* which means you can control who can act as the recovery user.

For example, by setting `CLICON_NACM_CREDENTIALS` to *except* the RESTCONF daemon can make backend calls posing as the recovery user, even though it runs as *wwwuser*.

Alternatively, `CLICON_NACM_CREDENTIALS` can be set to *exact* and the recovery user as *root*, in which case only a netconf or cli session running as root can make recovery operations.

10.4 10.4 Confirm-commit

Confirm as defined in RFC 6241 Sec 8.4 is enabled by:

```
<CLICON_FEATURE>ietf-netconf:confirmed-commit</CLICON_FEATURE>
```

Confirmed-commit adds the `<cancel-commit>` operation and more parameters to `<commit>`.

The "rollback" datastore is added and is used as a temporary revert datastore.

10.5 10.5 Callhome

With Clixon, you can make a solution following RFC 8071: NETCONF Call Home and RESTCONF Call Home over SSH as a utility using openssh.

The solution is built "around" Clixon meaning that Clixon itself is used as-is. This may be referred to as "external" callhome since it is done using external tools, not clixon itself. In contrast, rstconf call-home is "internal", see callhome section in *Restconf section*.

Other solutions are possible as well, especially on the client side, and a full system integration requires a callhome framework to determine when and how callhomes are made as well as addressing the security implications addressed by RFC 8071.

Overview of a callhome architecture with a device (where clixon resides) and a client:

```
   device/server                          client
+-----------------+  2b) tcp connect   +----------------------+
| 2a) callhome    | ----------------->  | 1c) callhome-client |
+-----------------+                     +----------------------+
        | 3)                                  ^  |
        v                                  1b)|  v
+-----------------+   4) ssh session   +----------------------+   5) stdio
|     sshd -i     | <----------------> | 1a)   ssh            |  <------  <rpc>...</rpc>"
+-----------------+                    |----------------------+
        | stdio
+-----------------+
| clixon_netconf  |
+-----------------+
        |
```

(continues on next page)

(continued from previous page)

```
+-----------------+
| clixon_backend  |
+-----------------+
```

Requirement for the netconf callhome solutions are openssh and openssh-server.

The steps to make a Netconf callhome is as follows:

1) Start the ssh client using `-o ProxyUseFdpass=yes -o ProxyCommand="callhome-client"`. Callhome-client listens on port 4334 for incoming TCP connections.
2) Start the callhome program on the server making tcp connect to client on port 4334 establishing a tcp stream with the client
3) The callhome program starts `sshd -i` using the established stream socket
4) The callhome-client returns with an open stream socket to the ssh client establishing an SSH stream to the server
5) Netconf messages are sent on stdin to the ssh client in turn using the established SSH stream and the Netconf subsystem to clixon, which returns a reply.

The callhome and callhome-client referred to above are implemented by the utility functions: `util/clixon_netconf_ssh_callhome` and `util/clixon_netconf_ssh_callhome_client`.

Example:

```
# Start ssh on client: bind to 1.2.3.4:4334 sending an rpc on stdin
client> echo '<?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:1.1</capability></capabilities></hello>]]>]]><rpc xmlns=
↪"urn:ietf:params:xml:ns:netconf:base:1.0"><get-config><source><candidate/></source></
↪get-config></rpc>]]>]]>' > msg
client> ssh -s -v -o ProxyUseFdpass=yes -o ProxyCommand="clixon_netconf_ssh_callhome_
↪client -a 1.2.3.4" . netconf < msg

# Start callhome on server: connect to 1.2.3.4:4334
server> sudo clixon_netconf_ssh_callhome -a 1.2.3.4

# Reply on client stdout: (skipping hello):
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><data/></rpc-reply>]]>]]>
```

The example is implemented as a regression test in `test/test_netconf_ssh_callhome.sh`

The RFC lists several security issues that need to be addressed in a solution, including "pinning" of host keys etc.

Note: Warning: there are security implications of using this example as noted in RFC 8071: NETCONF Call Home and RESTCONF Call Home

10.6 10.6 Internal NETCONF

Clixon uses NETCONF in the internal IPC protocol between its clients (cli/netconf/restconf) and the backend. This *internal* Netconf (IPC) is slightly different from regular Netconf:

- A different framing
- Netconf extentions

Note: The IPC is an internal interface, do not use externally

10.6.1 10.6.1 Framing

A fixed header using session id and message length before the netconf message:

```
struct clicon_msg {
   uint32_t    op_len;     /* length of whole message: body+header, network byte order.␣
↪*/
   uint32_t    op_id;      /* session-id. network byte order. */
   char        op_body[0]; /* rest of message, actual data */
};
```

The `session-id` is a number determined by the server. In the first hello, the client can assign zero, and assign the correct session-id in subsequent messages. The server hello contains the assigned session-id.

10.6.2 10.6.2 Extensions

The internal IPC protocol have a couple of attributes that are extensions to the Netconf protocol. These attributes are all in the `clixon-lib` namespace (`http://clicon.org/lib`)

- *content* - for `get` command with values "config", "nonconfig" or "all", to indicate which parts of state and config are requested. This option is taken from RESTCONF. Example:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <get cl:content="nonconfig" xmlns:cl="http://clicon.org/lib"/>
</rpc>
```

- *depth* - for `get` and `get-config` how deep a tree is requested. Also from RESTCONF. Example:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <get cl:depth="2" xmlns:cl="http://clicon.org/lib"/>
</rpc>
```

- *username* - for top-level `rpc` command. Indicates which user the client represents ("pseudo-user"). This is either the actual user logged in as the client (eg "peer-user") or can represent another user. The credentials mode determines the trust-level of the pseudo-username. Example:

```
<rpc username="root" xmlns:cl="http://clicon.org/lib" xmlns=
↪"urn:ietf:params:xml:ns:netconf:base:1.0">
   <close-session/>
</rpc>
```

- *autocommit* - for `edit-config`. If true, perform a `commit` operation immediately after an edit. If this fails, make a `discard` operation. Example:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <edit-config cl:autocommit="true" xmlns:cl="http://clicon.org/lib">
      <target><candidate/></target>
      <config>...</config>
   </edit-config>
</rpc>
```

- *copystartup* - for `edit-config` combined with autocommit. If true, copy the running db to the startup db after a commit. The combination with autocommit is the default for RESTCONF operations. Example:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <edit-config cl:autocommit="true" cl:copystartup="true" xmlns:cl="http://clicon.
↪org/lib">
      <target><candidate/></target>
      <config>...</config>
   </edit-config>
</rpc>
```

- *transport* - for `hello` from RFC 6022. Example:

```
<hello cl:transport="netconf" xmlns:cl="http://clicon.org/lib" xmlns=
↪"urn:ietf:params:xml:ns:netconf:base:1.0" >
```

- *source-host* - for `hello` from RFC 6022. Example:

```
<hello cl:source-host="10.10.0.42" xmlns:cl="http://clicon.org/lib" xmlns=
↪"urn:ietf:params:xml:ns:netconf:base:1.0" >
```

- *objectcreate* and *objectexisted* - in the data field of `edit-config` XML data tree. In the request set objectcreate to false/true whether an object should be created if it does not exist or not. If such a request exists, then the ok reply should contain "objectexists" to indicate whether the object existed or not (eg prior to the operation). The reason for this protocol is to implement some RESTCONF PATCH and PUT functionalities. Example:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <edit-config objectcreate="false"><target><candidate/></target>
      <config>
         <protocol objectcreate="true">tcp</protocol>
       </config>
   </edit-config>
</rpc>]]>]]>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <ok objectexisted="true"/>
</rpc-reply>]]>]]>
```

The reason for introducing the objectcreate/objectexisted attributes are as follows:

- RFC 8040 4.5 PUT: if the PUT request creates a new resource, a "201 Created" status-line is returned. If an existing resource is modified, a "204 No Content" status-line is returned.
- RFC 8040 4.6 PATCH: If the target resource instance does not exist, the server MUST NOT create it.

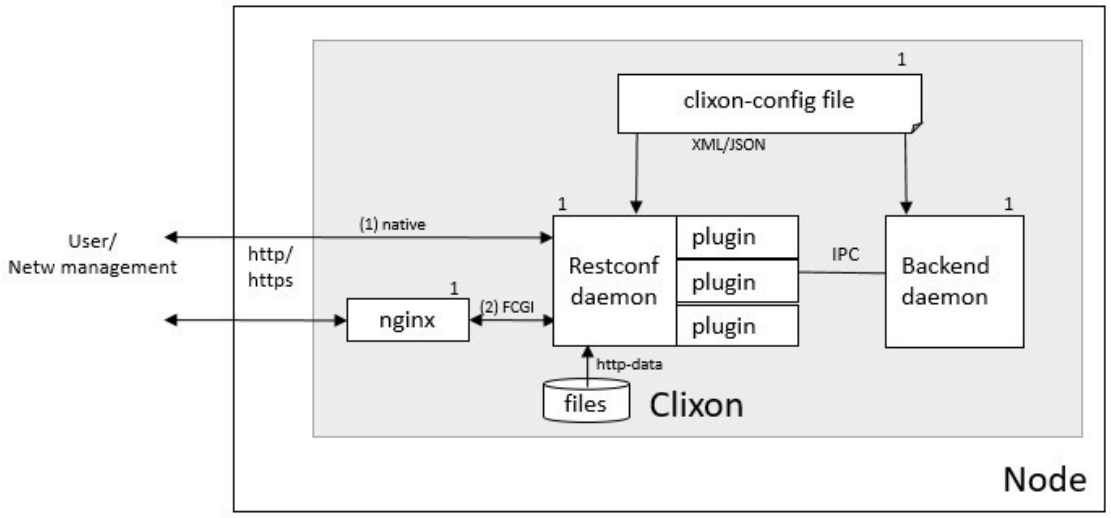
CHAPTER

ELEVEN

11 RESTCONF

Clixon supports two RESTCONF compile-time variants: *FCGI* and *Native*.

11.1 11.1 Architecture



The restconf deamon provides a http/https RESTCONF interface to the Clixon backend. It comes in two variants, as shown in the figure above:

1. Native http, which combines a HTTP and Restconf server. Further, HTTP configuration is made using Clixon.
2. A reverse proxy (such as NGINX) and FastCGI where web and restconf function is separated. NGINX is used to make all HTTP configuration.

The restconf daemon communicates with the backend using internal netconf over the `CLIXON_SOCK`. If FCGI is used, there is also a FCGI socket specified by `fcgi-socket` in `clixon-config/restconf`.

The restconf daemon reads its initial config options from the configuration file on startup. The native http variant can read config options from the backend as an alternative to reading everything from clixon options.

You can add plugins to the restconf daemon, where the primary usecase is authentication, using the `ca_auth` callback.

Note that there is some complexity in the configuration of the different variants of native Clixon restconf involving HTTP/1 vs HTTP/2, TLS vs plain HTTP, client cert vs basic authentication and external vs internal daemon start.

Further, ALPN is used to select http/1 or http/2 in HTTPS.

11.2 11.2 Installation

The RESTCONF daemon can be configured for compile-time (by autotools) as follows:

--disable-http1 Disable native http/1.1 (ie http/2 only)

--disable-nghttp2 Disable native http/2 using libnghttp2 (ie http/1 only)

--with-restconf=native RESTCONF using native http. (DEFAULT)

--with-restconf=fcgi RESTCONF using fcgi/ reverse proxy.

--without-restconf No RESTCONF

After that perform system-wide compilation:

```
make && sudo make install
```

11.3 11.3 Command-line options

The restconf daemon have the following command-line options:

-h	Help
-V	Show version and exit
-D <level>	Debug level
-f <file>	Clixon config file
-E <dir>	Extra configuration directory
-l <option>	Log on (s)yslog, std(e)rr, std(o)ut, (n)one or (f)ile. Syslog is default. If foreground, then syslog and stderr is default.
-C <format>	Dump configuration options on stdout after loading and quit. Format is one of xml|json|text
-p <dir>	Add Yang directory path (see CLICON_YANG_DIR)
-y <file>	Load yang spec file (override yang main module)
-a <family>	Internal backend socket family: UNIX|IPv4|IPv6
-u <path|addr>	Internal socket domain path or IP addr (see -a)
-r	Do not drop privileges if run as root
-W <user>	Run restconf daemon as this user, drop according to `CLICON_RESTCONF_PRIVILEGES`
-R <xml>	Restconf configuration in-line overriding config file
-o <option=value>	Give configuration option overriding config file (see clixon-config.yang)

Note that the restconf daemon started as root, drops privileges to *wwwuser*, unless the `-r` command-line option is used, or `CLICON_RESTCONF_PRIVILEGES` is defined.

11.4 11.4 Configuration options

The following RESTCONF configuration options can be defined in the clixon configuration file:

CLICON_RESTCONF_DIR
Location of restconf .so plugins. Load all .so plugins in this dir as restconf code plugins.

CLICON_RESTCONF_INSTALLDIR
Path to dir of clixon-restconf daemon binary as used by backend if started internally

CLICON_RESTCONF_STARTUP_DONTUPDATE
Disable automatic update of startup on restconf edit operations This is not according to standard RFC 8040 Sec 1.4.

CLICON_RESTCONF_USER
Run clixon_restconf daemon as this user, default is *www-data*.

CLICON_RESTCONF_PRIVILEGES
Restconf daemon drop privileges mode, one of: none, drop_perm, drop_temp

CLICON_RESTCONF_HTTP2_PLAIN
Enable plain (non-tls) HTTP/2.

CLICON_BACKEND_RESTCONF_PROCESS
Start restconf daemon internally from backend daemon. The restconf daemon reads its config from the backend running datastore.

CLICON_ANONYMOUS_USER
If RESTCONF authentication auth-type=none then use this user

CLICON_RESTCONF_API_ROOT
RESTCONF API root path as defined in RFC 8040, default is */restconf*

CLICON_NOALPN_DEFAULT
Fallback if no ALPN for https. valid values are "http/1.1" and "http/2"

More more documentation of the options, see the source YANG file.

11.5 11.5 Advanced config

Apart from options, there is also structured restconf data primarily for native mode encapsulated with `<restconf>...</restconf>` as defined in `clixon-restconf.yang`.

The first-level fields of the advanced restconf structure are the following:

enable
Enable the RESTCONF daemon. If disabled, the restconf daemon will not start

auth-type
Authentication method (see *auth types*)

debug
Enable debug

log-destination
Either syslog or file (/var/log/clixon_restconf.log)

pretty
Restconf vallues are pretty printed by default. Disable to turn this off

The advanced config can be given using three different methods

1. inline - as command-line option using -R
2. config-file - as part of the regular config file
3. datastore - committed in the regular running datastore

11.5.1 11.5.1 Inline

When starting the restconf daemon, structured data can be directly given as a command-line option:

```
-R <restconf xmlns="http://clicon.org/restconf"><enable>true</enable></restconf>
```

11.5.2 11.5.2 Config file

The restconf config can also be defined locally within the clixon config file, such as:

```
<CLICON_FEATURE>clixon-restconf:fcgi</CLICON_FEATURE>
<CLICON_BACKEND_RESTCONF_PROCESS>false</CLICON_BACKEND_RESTCONF_PROCESS>
<restconf>
    <enable>true</enable>
    <fcgi-socket>/wwwdata/restconf.sock</fcgi-socket>
 </restconf>
```

11.5.3 11.5.3 Datastore

Alternatively if CLICON_BACKEND_RESTCONF_PROCESS is set, the restconf configuration is:

```
<CLICON_FEATURE>clixon-restconf:fcgi</CLICON_FEATURE>
<CLICON_BACKEND_RESTCONF_PROCESS>false</CLICON_BACKEND_RESTCONF_PROCESS>
```

And the detailed restconf is defined in the regular running datastore by adding something like:

```
<restconf xmlns="http://clicon.org/restconf">
   <enable>true</enable>
   <fcgi-socket>/wwwdata/restconf.sock</fcgi-socket>
</restconf>
```

In the latter case, the restconf daemon reads its config from the running datastore on startup.

Note: If CLICON_BACKEND_RESTCONF_PROCESS is enabled, the restconf config must be in the regular datastore.

11.5.4 11.5.4 Features

The Restconf config has two features:

fcgi
The restconf server supports the fast-cgi reverse proxy mode. Set this if fcgi/nginx is used.

allow-auth-none
Authentication supports a *none* mode.

Example, add this in the config file to enable fcgi:

```
<clixon-config xmlns="http://clicon.org/config">
   ...
   <CLICON_FEATURE>clixon-restconf:fcgi</CLICON_FEATURE>
```

11.5.5 11.5.5 Auth types

The RESTCONF daemon uses the following authentication types:

none
Messages are not authenticated and set to the value of `CLICON_ANONYMOUS_USER`. A callback can revise this behavior. Note, must set *allow-auth-none* feature.

client-cert
Set to authenticated and extract the username from the SSL_CN parameter. A callback can revise this behavior.

user
User-defined behaviour as implemented by the *auth callback*. Typically done by basic auth, eg HTTP_AUTHORIZATION header, and verify password

11.5.6 11.5.6 FCGI mode

Applies if clixon is configured with `--with-restconf=fcgi`. Fcgi-specific config options are:

fcgi-socket
Path to FCGI unix socket. This path should be the same as specific in fcgi reverse proxy

Need also fcgi feature enabled: *features*

11.5.7 11.5.7 Native mode

Applies if clixon is configured with `--with-restconf=native`. Native specific config options are:

server-cert-path
Path to server certificate file

server-key-path
Path to server key file

server-ca-cert-path
Path to server CA cert file

socket
List of server sockets that the restconf daemon listens to with the following fields:

socket namespace
Network namespace

socket address
IP address to bind to

socket port
TCP port to bind to

socket ssl
If true: HTTPS; if false: HTTP protocol

11.5.8 11.5.8 Examples

Configure a single HTTP on port 80 in the default config file:

```
<clixon-config xmlns="http://clicon.org/config">
   <CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
   ...
   <restconf>
      <enable>true</enable>
      <auth-type>user</auth-type>
      <socket>
         <description>HTTP listen</description>
         <namespace>default</namespace>
         <address>0.0.0.0</address>
         <port>80</port>
         <ssl>false</ssl>
      </socket>
   </restconf>
</clixon-config>
```

Configure two HTTPS listeners in two different namespaces:

```
<restconf xmlns="https://clicon.org/restconf">
   <enable>true</enable>
   <auth-type>client-certificate</auth-type>
   <server-cert-path>/etc/ssl/certs/clixon-server-crt.pem</server-cert-path>
   <server-key-path>/etc/ssl/private/clixon-server-key.pem</server-key-path>
   <server-ca-cert-path>/etc/ssl/certs/clixon-ca_crt.pem</server-ca-cert-path>
   <socket>
      <description>HTTPS listen</description>
      <namespace>default</namespace>
      <address>0.0.0.0</address>
      <port>443</port>
      <ssl>true</ssl>
   </socket>
   <socket>
      <description>HTTPS listen on myns</description>
      <namespace>myns</namespace>
      <address>0.0.0.0</address>
      <port>443</port>
      <ssl>true</ssl>
   </socket>
</restconf>
```

11.5.9 11.5.9 SSL Certificates

If you use native RESTCONF you may want to have server/client certs. If you use FCGI, certs are configured according to the reverse proxy documentation, such as NGINX. The rest of this section applies to native restconf only.

If you already have certified server certs, ensure `CLICON_SSL_SERVER_CERT` and `CLICON_SSL_SERVER_KEY` points to them.

If you do not have them, you can generate self-signed certs, for example as follows:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout /etc/ssl/private/clixon-server-key.pem␣
↪-out /etc/ssl/certs/clixon-server-crt.pem -days 365
```

You can also generate client certs (not shown here) using `CLICON_SSL_CA_CERT`. Example using client certs and curl for client *andy*:

```
curl -Ssik --key andy.key --cert andy.crt -X GET https://localhost/restconf/data/
↪example:x
```

11.6 11.6 Starting

You can start the RESTCONF daemon in several ways:

1. *systemd* , externally started
2. *internally* using the *process-control* RPC (see below)
3. *docker* mechanisms, see the docker container docs

11.6.1 11.6.1 Start with Systemd

The Restconf service can be installed at, for example, /etc/systemd/system/example_restconf.service:

```
[Unit]
Description=Starts and stops an example clixon restconf service on this system
Wants=example.service
After=example.service
[Service]
Type=simple
User=root
Restart=on-failure
ExecStart=/usr/local/sbin/clixon_restconf -f /usr/local/etc/example.xml
[Install]
WantedBy=multi-user.target
```

11.6.2 11.6.2 Internal start

For starting restconf internally, you need to enable `CLICON_BACKEND_RESTCONF_PROCESS` option. See Section *datastore*.

Thereafter, you can either use the `clixon-restconf.yang` configuration or use the `clixon-lib.yang` process control RPC:s to start/stop/restart the daemon or query status.

The algorithm for starting and stopping the clixon-restconf internally is as follows:

1. on RPC start, if enable is true, start the service, if false, error or ignore it
2. on RPC stop, stop the service
3. on backend start make the state as configured
4. on enable change, make the state as configured

Example 1, using netconf *edit-config* to start the process:

```
<?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:1.1</capability></capabilities>
</hello>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
   <edit-config>
      <default-operation>merge</default-operation>
      <target><candidate/></target>
      <config>
         <restconf xmlns="http://clicon.org/restconf">
            <enable>true</enable>
         </restconf>
      </config>
   </edit-config
</rpc>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
   <ok/>
</rpc-reply>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="11">
   <commit/>
  </rpc>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
   <ok/>
</rpc-reply>
```

Example 2, using netconf RPC to restart the process:

```
<?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:1.1</capability></capabilities>
</hello>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
   <process-control xmlns="http://clicon.org/lib">
      <name>restconf</name>
      <operation>restart</operation>
   </process-control>
</rpc>
```

(continues on next page)

(continued from previous page)

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
   <pid xmlns="http://clicon.org/lib">1029</pid>
</rpc-reply>
```

Note that the backend daemon must run as root (no lowering of privileges) to use this feature.

11.7 11.7 Plugin callbacks

Restconf plugins implement callbacks, some are same as for *backend plugins*.

init
: Clixon plugin init function, called immediately after plugin is loaded into the restconf daemon.

start
: Called when application is started and initialization is complete, and after drop privileges.

exit
: Called just before plugin is unloaded

extension
: Called at parsing of yang modules containing an extension statement.

auth
: See *auth callback*

11.7.1 11.7.1 Auth callback

The role of the authentication callback is, given a message (its headers) and authentication type, determine if the message passes authentication and return an associated user.

The auth callback is invoked after incoming processing, including cert validation, if any, but before relaying the message to the backend for NACM checks and datastore processing.

If the message is not authenticated, an error message is returned with tag: *access denied* and HTTP error code *401 Unauthorized*.

There are default handlers for TLS client certs and for "none" authentication. But other variants, such as http basic authentication, oauth2 or the remapping of client certs to NACM usernames, can be implemented by this callback

If the message is authenticated, a user is associated with the message. This user can be derived from the headers or mapped in an application-dependent way. This user is used internally in Clixon and sent via the IPC protocol to the backend where it may be used for NACM authorization.

The signature of the auth callback is as follows:

```
int ca_auth(clixon_handle h, void *req, clixon_auth_type_t auth_type, char **authp);
```

where:

h
: Clixon handle

req
: Per-message request www handle to use with restconf_api.h

auth-type
: Specifies how the authentication is made and what default value

authp
NULL if credentials failed, otherwise malloced string of authentoicated user

The return value is one of:

- -1: Fatal error, close socket
- 0: Ignore, undecided, not handled, same as no callback. Fallback to default handler.
- 1: OK see authp parameter whether the result is authenticated or not, and the associated user.

If there are multiple callbacks, the first result which is not "ignore" is returned. This is to allow for different callbacks registering different classes, or grouping of authentication.

The main example contains example code.

11.8 11.8 FCGI

This section describes the RESTCONF FCGI mode using NGINX.

You need to configure the following:

1. Configure clixon with `--with-restconf=fcgi`
2. Restconf config in the Clixon config file
3. Reverse proxy configuration
4. Start the restconf daemon (see *starting*)

11.8.1 11.8.1 Restconf config

The restconf daemon can be started in several ways as described in Section *auth types*. In all cases however, the configuration is simpler than in native mode. For example:

```
<clixon-config xmlns="http://clicon.org/config">
   ...
   <CLICON_FEATURE>clixon-restconf:fcgi</CLICON_FEATURE>
   <restconf>
      <enable>true</enable>
      <fcgi-socket>/wwwdata/restconf.sock</fcgi-socket>
   </restconf>
</clixon-config>
```

11.8.2 11.8.2 Reverse proxy config

If you use FCGI, you need to configure a reverse-proxy, such as NGINX. A typical configuration is as follows:

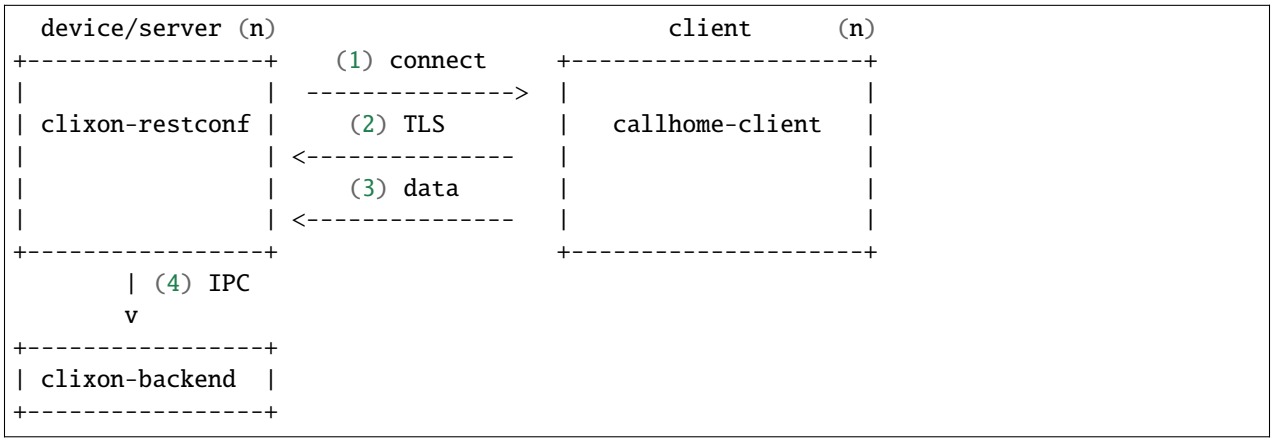
```
server {
  ...
  location / {
    fastcgi_pass unix:/www-data/fastcgi_restconf.sock;
    include fastcgi_params;
  }
}
```

where `fastcgi_pass` setting should match by `fcgi-socket` in `clixon-config/restconf`.

11.9 11.9 Callhome

Clixon supports RESTCONF callhome according to RFC 8071: NETCONF Call Home and RESTCONF Call Home using native RESTCONF and TLS and server/client certs.

11.9.1 11.9.1 Overview



The operation of RESTCONF callhome is as follows:

1. The RESTCONF server initiates a TCP connection to a client, either persistently or periodically
2. The client sets up a TLS connection to the server using the existing TCP session
3. The client sends data as HTTP requests over TLS to the server
4. The RESTCONF server receives data, authenticates the client-cert, transforms the request to NETCONF and sends it internally to the clicon backend.
5. Status replies are returned to the client

Note: Clixon does not implement client-side call-home functionality, only server-side

11.9.2 11.9.2 Callback clients

A server may configure multiple HTTP callback clients, for fault-tolerance purposes, for example.

A callback "controller" client typically serves multiple servers.

Bootstrap controller

A typical callback client scenario is a bootstrap controller. The HTTP requests by the client are stored waiting for an inital connect from a minimally deployed server. The controller may then store pre-configured configurations. In such a scenario, the server needs at least the following initial information:

1. A callback IP address
2. A server cert
3. A CA to validate the client cert OR a list of exactly matching client-certs

Once the controller accepts a connection from a server, it may send RESTCONF requests over HTTP and fully configure the new server.

Such a controller can also be used for more "intelligent" configuration as well, such as setting up tunnels or other configurations spanning multiple servers.

The controller may also provide an interactive CLI or GUI for example once a connection is established.

11.9.3 11.9.3 Description

The callhome function is "internal" in the sense that it is integrated in Clixon and uses the openssl lib and extends the regular "listen" RESTCONF functionality.

The callhome mechanism is a server-side implementation. There is an example client-side implementation (*util/clixon_restconf_callhome_client.c*) which is not a part of the actual Clixon code. A user needs to write a client to use this functionality.

The existing clixon-restconf YANG has been extended to support callhome. The ietf restconf server draft (https://datatracker.ietf.org/doc/html/draft-ietf-netconf-restconf-client-server-26) which is used as a basis for the extensions. While not complying to the draft's structure, the YANG fields covering callhome are similar. Please see the draft for detailed description of scenarios and configuration fields.

The callhome features include:

- *Persistent* and *periodic* connection types, ie continuous callhome attempts, or at specific time intervals.
- Periodic connections support *idle-timeout*, ie close the TCP connection if no traffic after timeout.
- *Max-attempts* reconnect strategy, ie how many times to retry a connect attempt before timeout

11.9.4 11.9.4 Setup

A callhome session is setup by adding a `call-home` section to a native RESTCONF socket declaration. For example:

```
<restconf xmlns="https://clicon.org/restconf">
 <enable>true</enable>
 <auth-type>client-certificate</auth-type>
 <server-cert-path>/etc/ssl/certs/clixon-server-crt.pem</server-cert-path>
 <server-key-path>/etc/ssl/private/clixon-server-key.pem</server-key-path>
 <server-ca-cert-path>/etc/ssl/certs/clixon-ca_crt.pem</server-ca-cert-path>
 <socket>
    <description>callhome session</description>
    <namespace>default</namespace>
    <address>12.13.14.15</address>
    <port>4336</port>
    <ssl>true</ssl>
```

(continues on next page)

(continued from previous page)

```
    <call-home>
      <!-- ... call-home section ... -->
    </call-home>
 </socket>
```

Some notes for callhome sockets:

1. The `address` field denotes a remote client, not the server which is the case for "listen" sockets.
2. The default `port` for RESTCONF callhome is *4336*
3. A callhome socket must have `ssl` enabled
4. Client certs must be used as `auth-type`
5. You can mix regular "listen" sockets with "callhome" sockets.
6. You can have multiple (concurrent) callhome sockets.

11.9.5 11.9.5 Persistent connection

If the callhome session is persistent, the server tries to hold the connection open at all times. The default re-connect strategy is 1 second.

Example socket configuration:

```
<call-home>
   <connection-type>
      <persistent/>
   </connection-type>
</call-home>
```

11.9.6 11.9.6 Periodic connection

Periodic call-home sessions try to establish a callhome connections at regular intervals, such as once a minute, or once a day.

Example periodic configuration:

```
<call-home>
   <connection-type>
      <periodic>
         <period>3600</period>
         <idle-timeout>60</idle-timeout>
       </periodic>
   </connection-type>
   <reconnect-strategy>
      <max-attempts>3</max-attempts>
   </reconnect-strategy>
</call-home>
```

Notes:

1. The *period* is in seconds, while the draft uses minutes. The example therefore shows a period of one hour.
2. The *idle-timeout* field means that the TCP session is closed by the server if no data is sent in 1 minute.

3. The *max-attempts* setting means that every start of period the server makes several attempts to reconnect. If all attempts fail, it waits another period before re-trying.
4. If the connection is active when the new period starts, no reconnect attempt is made.

11.10 11.10 HTTP data

As an extension to the native restconf implementation, Clixon provides a limited http-data server for displaying web pages. To use the http-data feature, you need to configure clixon with `--with-restconf=native` and also enable the `http-data` feature as described below.

Note: The http data feature is limited and can only be used for very simple web content

11.10.1 11.10.1 Configuration options

First, you also need to define `<CLICON_FEATURE>clixon-restconf:http-data</CLICON_FEATURE>` to enable http-data.

The following configuration options can be defined in the clixon configuration file:

CLICON_HTTP_DATA_ROOT
: Directory in the local file system where http-data files are searched for. Soft links, `..`, ~ etc are not followed. Default is `/var/www`.

CLICON_HTTP_DATA_PATH
: Prefix to match with URI to match for http-data. This path is appended to `CLICON_HTTP_DATA_ROOT` to find a matching file. Default is `/`. Note that the restconf match prefix is `/restconf`.

Example

The following is an example of a config file for setting up an http-data service at `/var/www/data`:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_FEATURE>clixon-restconf:http-data</CLICON_FEATURE>
  <CLICON_HTTP_DATA_ROOT>/var/www</CLICON_HTTP_DATA_ROOT>
  <CLICON_HTTP_DATA_PATH>/data</CLICON_HTTP_DATA_PATH>
  ...
```

An example curl call could be:

```
curl -X GET -H 'Accept: text/html' http://localhost/data/
```

The call will retrieve the file at `/var/www/data/index.hmtl`.

11.10.2 11.10.2 Features and limitation

The http server is very limited in functionality:

- No dynamic pages, ie backend scripts, only static pages are supported.
- Operations are limited to GET, HEAD, and OPTIONS
- Query parameters are not supported
- Indata is not supported
- Supported media is: html, css, js, fonts, and some images. All other content is default *octet-stream*

All applicable features of the native restconf implementation are available for the http-data as well. This includes http/1 and http/2 and TLS using openssl.

There is support for URI path internal redirect to a file called *index.html*. This can be changed by compile-time option `HTTP_DATA_INTERNAL_REDIRECT`.

11.11 11.11 RESTCONF streams

Clixon has an experimental RESTCONF event stream implementations following RFC8040 Section 6 using Server-Sent Events (SSE). Currently this is implemented in FCGI/Nginx only (not native).

Note: RESTCONF streams are experimental and only implemented for FCGI.

Example: set the Clixon configuration options:

```
<CLICON_STREAM_PATH>streams</CLICON_STREAM_PATH>
<CLICON_STREAM_URL>https://example.com</CLICON_STREAM_URL>
<CLICON_STREAM_RETENTION>3600</CLICON_STREAM_RETENTION>
```

In this example, the stream `example` is accessed with `https://example.com/streams/example`.

Clixon defines an internal in-memory (not persistent) replay function controlled by the configure option above. In this example, the retention is configured to 1 hour, i.e., the stream replay function will only save timeseries one hour, but if the restconf daemon is restarted, the history will be lost.

In the Nginx configuration, add the following to extend the nginx configuration file with the following statements (for example):

```
location /streams {
    fastcgi_pass unix:/www-data/fastcgi_restconf.sock;
    include fastcgi_params;
    proxy_http_version 1.1;
    proxy_set_header Connection "";
}
```

An example of a stream access is as follows:

```
curl -H "Accept: text/event-stream" -s -X GET http://localhost/streams/EXAMPLE
data: <notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0"><eventTime>
↪2018-11-04T14:47:11.373124</eventTime><event><event-class>fault</event-class>
↪<reportingEntity><card>Ethernet0</card></reportingEntity><severity>major</severity></
```

(continues on next page)

(continued from previous page)

```
↪event></notification>
data: <notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0"><eventTime>
↪2018-11-04T14:47:16.375265</eventTime><event><event-class>fault</event-class>
↪<reportingEntity><card>Ethernet0</card></reportingEntity><severity>major</severity></
↪event></notification>
```

You can also specify start and stop time. Start-time enables replay of existing samples, while stop-time is used both for replay, but also for stopping a stream at some future time:

```
curl -H "Accept: text/event-stream" -s -X GET http://localhost/streams/EXAMPLE?start-
↪time=2014-10-25T10:02:00&stop-time=2014-10-25T12:31:00
```

11.11.1 11.11.1 Fcgi stream options

The following options apply only for fcgi mode and notification streams:

CLICON_STREAM_DISCOVERY_RFC8040
: Enable monitoring information for the RESTCONF protocol from RFC 8040 (only fcgi)

CLICON_STREAM_PATH
: Stream path appended to CLICON_STREAM_URL to form stream subscription URL (only fcgi)

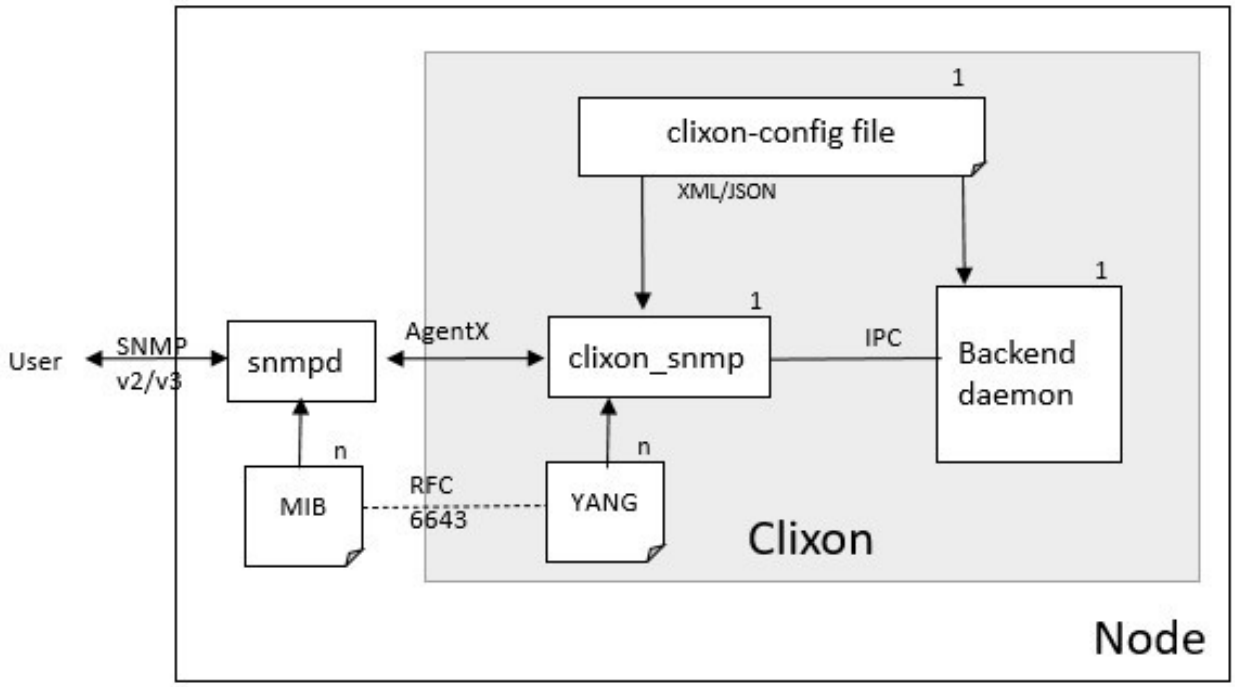
CHAPTER

TWELVE

12 SNMP

Clixon supports SNMP for retreiving and setting values via netsnmp using a MIB-YANG mapping defined in RFC6643.

12.1 12.1 Architecture



The SNMP frontend acts as an intermediate daemon between the Net-SNMP daemon (snmpd) and the Clixon backend. Clixon-snmp communicates over the AgentX protocol to snmpd typically via a UNIX socket, and over the internal IPC protocol to the Clixon backend.

Clixon-snmp implements RFC 6643 *Translation of Structure of Management Information Version 2 (SMIv2) MIB Modules to YANG Modules*. The RFC defines how a MIB is translated to YANG using extensions that define a mapping between YANG statements and SMI object IDs and types in the MIB.

In principle, it is also possible to construct a MIB from YANG using the same method, although this is more limited and may involve manual work.

A user can then communicate with snmpd using any of the SNMP v2/v3 tools, such as *snmpget*, *snmpwalk* and others.

Note: SNMP support is introduced in Clixon version 5.8

12.2 12.2 Configuration

12.2.1 12.2.1 Net-SNMP

Note: Use Net-SNMP version 5.9 or later

To set up AgentX communication between `clixon_snmp` and `snmpd` a Unix or TCP socket is configured. This socket is also configured in Clixon (see below). An example */etc/snmp/snmpd.conf* is as follows:

```
master       agentx
agentaddress 127.0.0.1,[::1]
rwcommunity  public localhost
agentXSocket unix:/var/run/snmp.sock
agentxperms  777 777
```

It is necessary to ensure snmpd does *not* to load modules implemented by Clixon. For example, if Clixon implements the IF-MIB and system MIBs, snmpd should not load those modules. This can be done using the "-I" flag and prepending a "-" before each module:

```
-I -ifTable -I -system_mib -I -sysORTable
```

Further, Clixon itself does not start netsnmp itself, you need to ensure that netsnmp is running when clixon_snmp is started. Likewise, if snmpd is restarted, clixon_snmp must also be restarted.

Note: Net-snmp must be started via systemd or some other external mechanism before clixon_snmp is started.

12.2.2 12.2.2 Clixon

To build the snmp support, netsnmp is enabled at configure time. Two configure options are added for SNMP:

- `--enable-netsnmp` Enable SNMP support.
- `--with-mib-generated-yang-dir` For tests: Directory of generated YANG specs (default: $prefix/share/mibyang)

Then type "make" to build the "clixon_snmp" executable and "make install" to install.

12.2.3 12.2.3 clixon_snmp command line options

```
$ clixon_snmp -h
usage:clixon_snmp
where options are
     -h              Help
     -V              Show version and exit
     -D <level>      Debug level
     -f <file>       Configuration file (mandatory)
     -l (e|o|s|f<file>) Log on std(e)rr, std(o)ut, (s)yslog(default), (f)ile
     -C <format>     Dump configuration options on stdout after loading and exit. Format␣
↪is one of xml|json|text
     -z              Kill other clixon_snmp daemon and exit
     -o "<option>=<value>"   Give configuration option overriding config file (see␣
↪clixon-config.yang)
```

12.2.4 12.2.4 clixon_snmp configuration

There are two SNMP related options in the Clixon configuration:

CLICON_SNMP_AGENT_SOCK
: String description of the AgentX socket that clixon_snmp listens to.

CLICON_SNMP_MIB
: Names of MIBs that are used by clixon_snmp.

Example:

```
<CLICON_SNMP_AGENT_SOCK>unix:/var/run/snmp.sock</CLICON_SNMP_AGENT_SOCK>
<CLICON_SNMP_MIB>IF-MIB</CLICON_SNMP_MIB>
```

Note that the socket `/var/run/snmp.sock` is the same as configured in "snmpd.conf" above.

12.3 12.3 MIB mapping

Clixon registers MIBs with netsnmp by using YANG specifications. To achieve this, the MIB is first converted (according to RFC6643) to YANG format.

12.3.1 12.3.1 Generating YANG

MIB to YANG conversion can be done using the `smidump` tool, version 0.5 or later. Manual mapping is also possible. In Debian smidump is available in the package "smitools". You may also find existing repos with converted MIBs.

To convert a MIB to YANG, invoke `smidump` with the "-f yang" flag and point it to a MIB. MIBs will usually be in the directory "/usr/share/snmp/mibs/":

```
$ smidump -f yang /usr/share/snmp/mibs/IF-MIB.txt > IF-MIB.yang
```

Note: smidump 0.5 or later must be used

Once a MIB is converted to YANG, two things should be done:

1) The YANG is registered as an SNMP module using the `CLICON_SNMP_MIB` configuration option
2) The YANG file must be placed so that it can be found using the regular Clixon YANG finding mechanism, as described in *Finding YANG files*

12.3.2 12.3.2 Config vs state

By default, all RFC6643 mappings are `config false`, ie, no configuration data.

To change to configuration data, a deviation statement is made as the following example illustrates:

```
  deviation "/clixon-types:CLIXON-TYPES-MIB" {
   deviate replace {
      config true;
   }
}
```

For more info, see Section 11 in RFC 6643.

12.3.3 12.3.3 Types

Scalar types are mapped between SMI and YANG types using RFC6643. All types as used by IF-MIB, System and Entity MIBs are supported.

12.3.4 12.3.4 Tables

SNMP tables are supported for config and state and are mapped to YANG lists. It is possible to get and set individual values either via the SNMP API, or via any of the other CLIXON frontends.

Table indexes can be integers and non-integers. Multiple table indexes are supported.

As an implementation detail, Clixon uses the *table* abstraction in the netsnmp agent library, not *table-data* or *table-instance*.

12.3.5 12.3.5 RowStatus

Clixon supports SMIv2 RowStatus for table handling. Where RowStatus is used, the status of the row is returned and set to either *active*, *notInService* or *notReady*.

When writing the status of the row can be set to either *createAndGo*, *createAndWait*, *active* or *destroy*.

The *rowstatus* firled itself and all row values in *createAndWait* mode uses an internal cache which is held in memory by the clixon snmp agent. This internal cache is flushed to Clixon when setting a row to *active*, like a "pre-commit phase". When *clixon_snmp* is restarted, the cache is cleared.

CHAPTER

THIRTEEN

13 YANG

This chapter describes some aspects of the YANG implementation in Clixon. Regarding standard compliance, see *Standards*.

13.1 13.1 Leafrefs

Some notes on YANG leafref implementation in Clixon, especially as used in openconfig modules, which rely heavily on leafrefs.

Typically, a YANG leafref declaration looks something like this:

```
container c {
  leaf x {
    type leafref {
      path "../config/name";   /* "deferring node" */
      require-instance <bool>; /* determines existing deferred node */
    }
  }
  container config {
    leaf name {
      type unit32;             /* "deferred node" */
    }
  }
}
```

This YANG example is typical of Openconfig lists defined in the openconfig modeling, where a key leaf references a "config" node further down in the tree.

Other typical uses is where the path is an absolute path, such as eg `path "/network-instances/network-instance/config/name";`

13.1.1 13.1.1 Types

Consider the YANG example above, the type of `x` is the deferred node:s, in this example `uint32`. The validation/commit process, as well as the autocli type system and completion handles accordingly.

For example, if the deferred node is a more complex type such as identityref with options "a, b", the completion of "x" will show the options "a,b".

13.1.2 13.1.2 Require-instance

Assume the yang above have the following two XML trees:

```
<c>
  <x>foo</x>
</c>
```

and:

```
<c>
  <x>foo</x>
  <config>
    <name>foo</name>
  </config>
</c>
```

The validity of the trees is controlled by the require-instance property . According to this semantics:

- If require-instance is false, both trees above are valid,
- If require-instance is true(or not present), the upper tree is invalid and the lower is valid

In most models defined by openconfig and ietf, require-instance is typically false.

13.2 13.2 YANG Library

Clixon partially supports YANG library RFC 8525 that provides information about YANG modules and datastores,

The following configure options are associated to the YANG library

CLICON_YANG_LIBRARY
: Enable YANG library support as state data according to RFC8525. Default: `true`

CLICON_MODULE_SET_ID
: Contains a server-specific identifier representing the current set of modules and submodules.

CLICON_XMLDB_MODSTATE
: Tag datastores with RFC 8525 YANG Module Library info. See *datastore* for details on how to tag datastores with Module-set info.

The module-set of RFC8525 can be retrieved using NETCONF get or RESTCONF GET as operational data. The fields that are supported are the following:

- Content-id of the whole module-set
- Name of each module
- Namespace
- Revision
- Feature
- Submodules

The following fields are not supported

- import-only-module
- deviation

- schema
- datastore

13.2.1 13.2.1 Example

An example of a NETCONF `get` reply with module-state data of the main example is the following:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="42">
  <data>
    <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
      <module-set>
        <name>default</name>
        <module>
          <name>clixon-autocli</name>
          <revision>2022-02-11</revision>
          <namespace>http://clicon.org/autocli</namespace>
        </module>
        <module>
          <name>clixon-example</name>
          <revision>2020-12-01</revision>
          <namespace>urn:example:clixon</namespace>
        </module>
        ...
      </module-set>
    </yang-library>
  </data>
</rpc-reply>
```

13.3 13.3 Extensions

Clixon implements YANG extensions. There are several uses, but one is to "annotate" a YANG specification with application-specific data that can be used in plugin code for some reason.

An extension with an argument is introduced in YANG as follows:

```
module example-lib {
   namespace "urn:example:lib";
   extension mymode {
      argument annotation;
   }
```

Such an extension can then be used in YANG declarations in two ways, either *inline* or *augmented*.

An inlined extension is useful in a YANG module that the designer has control over and can add extension reference directly in the YANG specification.

Assume for example that an interface declaration is extended with the extension declared above, as follows:

```
module my-interface {
  import example-lib{
    prefix exl;
  }
```

(continues on next page)

(continued from previous page)

```
  container "interfaces" {
    list "interface" {
      exl:mymode "my-interface";
      ...
```

If you instead use an external YANG, where you cannot edit the YANG itself, you can use augmentation instead, as follows:

```
module my-augments {
 import example-lib{
    prefix exl;
 }
 import ietf-interfaces{
    prefix if;
 }
 augment "/if:interfaces/if:interface"{
    exl:mymode "my-interface";
 }
 ...
```

When this is done, it is possible to access the extension value in plugin code and use that value to perform application-specific actions. For example, assume an XML interface object `x` retrieve the annotation argument:

```
char      *value = NULL;
int        exist = 0;
yang_stmt *y = xml_spec(x);

if (yang_extension_value(y, "mymode", "urn:example:lib", &exist, &value) < 0)
   err;
if (exist){
   // use extension value
   if (strcmp(value, "my-interface") == 0)
      ...
```

A more advanced usage is possible via an extension callback (`ca_callback`) which is defined for backend, cli, netconf and restconf plugins. This allows for advanced YANG transformations. Please consult the main example to see how this could be done.

13.4 13.4 Unique

The YANG unique statement is described in Section 7.8.3 of RFC 7950. However, the RFC is somewhat vague in the descriptions of its arguments.

Clixon therefore supports two simultaneous distinct cases: multiple direct children and single descendants

13.4.1 13.4.1 Multiple direct children

This is examplified in the RFC, such as:

```
list server {
  key "name";
  unique "ip port";
  leaf ip...
  leaf port...
```

where `ip` and `port` are direct children of `server` and the uniquess applies to their combination in all list instances.

13.4.2 13.4.2 Single descendants

The RFC says:

> schema node identifiers, which MUST be given in the descendant form

This does not exclude more elaborate schema nodes than direct children but are not explicitly allowed.

Therefore, Clixon also supports a single advanced schema node id. Such a schema node id may define a set of leafs. The uniqueness is then validated against all instances, such as for example:

```
list server {
  key "name";
  unique c/inner/value;
  container c {
     list inner {
        leaf value...
```

However, only a *single* such argument is allowed. The reason is that such a schema node may potentially refer to a set of instances (not just one) and the semantics of a combination of multiple such ids is unclear.

13.5 13.5 If-feature and anydata

The YANG if-feature statement is described in Section 7.20.2 of RFC 7950. The RFC states that:

> Definitions tagged with "if-feature" are ignored when the server does not support that feature.

This is implemented by doing the following to disabled YANG nodes:

(1) Configuration data nodes are replaced locally to a single ANYDATA data. This means that XML derived from disabled features are accepted but no validation is possible.

(2) Other YANG nodes, such as RPCs or state data are removed.

Example, assume the following YANG:

```
container c{
   if-feature A;
   leaf b {
      type string;
   }
}
rpc r {
      input {
          leaf x {
              if-feature A;
              type string;
          }
      }
}
```

If feature `A` is NOT enabled, the YANG is transformed to:

```
anydata c{
}
rpc r {
      input {
      }
}
```

The following config option is related:

CLICON_YANG_UNKNOWN_ANYDATA
: Treat unknown XML/JSON nodes as anydata when loading from startup db.

13.6 13.6 Schema mounts

Clixon implements Yang schema mounts as defined in: RFC 8528: YANG Schema Mount with the following restrictions:

1. A YANG mount-point can only be defined in a *presence container*.
2. Only *inline* mount-points are supported
3. *config false* mount-points are not supported

13.6.1 13.6.1 Configuration

The following configure options are associated to mount-points:

CLICON_YANG_SCHEMA_MOUNT
: Enable YANG library support as state data according to RFC8525. Should be set to: `true`

13.6.2 13.6.2 YANG

Mount-points are enabled by importing *ietf-yang-schema-mount* and then apply the *mount-point* extension at a presence container. Example:

```
module mymod {
   namespace "urn:example:my";
   ...
   import ietf-yang-schema-mount {
      prefix yangmnt;
   }
   list mylist {
      key name;
      leaf name{
         type string;
      }
      container myroot{
         presence "Otherwise root is not visible";
         yangmnt:mount-point "mylabel"{
            description "Root for other yang models";
         }
      }
   }
}
```

Once declared in the YANG schema, moint-points will appear dynamically in the data as they are added. For example, if a NETCONF <*edit-config*> adds the *myroot* container above, it will be recognized as a mount-point and populated with another set of YANG modules than the top-level.

Populating a moint-point with YANG schemas is made by an application-dependent callback as described in Section *Mount callback*.

13.6.3 13.6.3 State

The moint-points appear in the state-data and can be retrieved using NETCONF get. The data appears in two places.

First, on the top-level *schema-mounts*:

```
<schema-mounts xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-schema-mount">
   <mount-point>
      <module>mymod</module>
      <label>mylabel</label>
      <config>true</config>
      <inline/>
   </mount-point>
</schema-mounts>
```

Second, at the mount-point level for all dynamically added moint-points:

```
<mylist xmlns="urn:example:my">
   <name>x</name>
   <myroot>
      <yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
         <module-set>
```

(continues on next page)

(continued from previous page)

```
            <name>mylabel</name>
            <module>
               <name>clixon-mount1</name>
               <namespace>urn:example:mount1</namespace>
               <revision>2023-05-01</revision>
            </module>
         </module-set>
      </yang-library>
   </myroot>
</mylist>
```

In this example, there is one dynamically created moint-point in the list *x* where the single YANG module *clixon-mount1* is mounted.

13.6.4 13.6.4 Mount callback

Mount-points need to be populated with YANG schemas. This is done by defining the *ca_yang_mount* callback. The following example illustrates how this is done in a C plugin:

```
static clixon_plugin_api api = {
    ...
    .ca_yang_mount=example_mount,
```

As input the callback takes the XML mount-point, and as output a yang-lib module-set tree. It also provides how to validate the YANG schemas and whether it is read-only or read-write:

```
int
main_yang_mount(clixon_handle   h,
                cxobj          *xt,
                int            *config,
                validate_level *vl,
                cxobj         **yanglib)
```

For example, the callback could return something like:

```
<yang-library xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
   <module-set>
      <name>mount</name>
      <module>
         <name>clixon-mount1</name>
         <namespace>urn:example:mount1</namespace>
         <revision>2023-05-01</revision>
      </module>
   </module-set>
</yang-library>
```

Clixon calls this callback when needed, such as when a new mount-point is created.

13.6.5 13.6.5 CLI

It is possible to extend the Autocli with mount-points. However, it is application-dependent. For the interested user, the Clixon controller has an adapted autocli for mount-points.

CHAPTER

FOURTEEN

14 USECASES

This section contains usecases which illustrate the flow of data from a user via Clixon frontends, backend to the underlying system and back.

14.1 14.1 CLI read

The first usecase illustrates how a retrieval of a configured value from the system is made.

1. The user makes a *show config* call using the hello world example(see *Quickstart section*). In the following examples uses *text* as modifier, and filters on *hello* top-level symbol:

```
cli> show configuration text hello
hello world;
```

2. The CLI string *show configuration text hello* is translated to internal NETCONF and sent to the backend:

```
<?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:1.1</capability></capabilities>
</hello>]]>]]>
<rpc username="myuser" xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source><candidate/></source>
    <nc:filter nc:type="xpath" nc:select="hello" xmlns="urn:example:hello"/>
  </get-config>
</rpc>
```

3. The backend receives the internal Netconf message, reads the *running* datastore and filters the output according to the XPath expression.

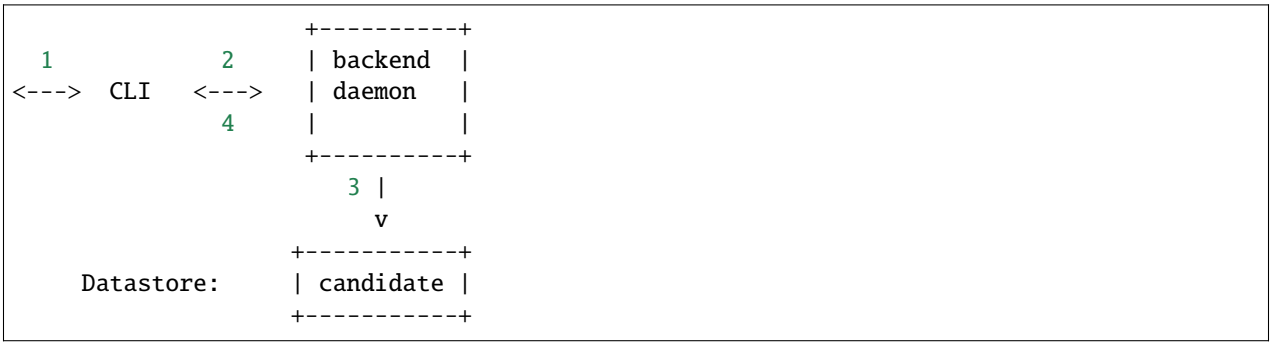
4. The backend returns the filtered output to the client:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <hello xmlns="urn:example:hello">
      <world/>
    </hello>
  </data>
</rpc-reply>
```

5. The CLI client translates the netconf to "text" output: *hello world;*

The user can also retrieve state data. Instead of reading from the running datastore, the backend reads state data either from a plugin, or from itself (if backend internal).

14.2 14.2 CLI write



The figure illustrates the way messages flow through the system. The numbers illustrate the enumeration below.

When setting a config value, the candidate datastore is modified and the committed to running which triggers a plugin commit transaction:

1. CLI example command:

```
cli> set hello world
cli>
```

2. Internal netconf containing a "replace" operation:

```
<?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:1.1</capability></capabilities>
</hello>]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:nc=
↪"urn:ietf:params:xml:ns:netconf:base:1.0" username="clicon">
  <edit-config>
    <target><candidate/></target>
    <default-operation>none</default-operation>
    <config>
      <hello xmlns="urn:example:hello">
        <world nc:operation="replace"/>
      </hello>
    </config>
```

(continues on next page)

(continued from previous page)

```
  </edit-config>
</rpc>
```

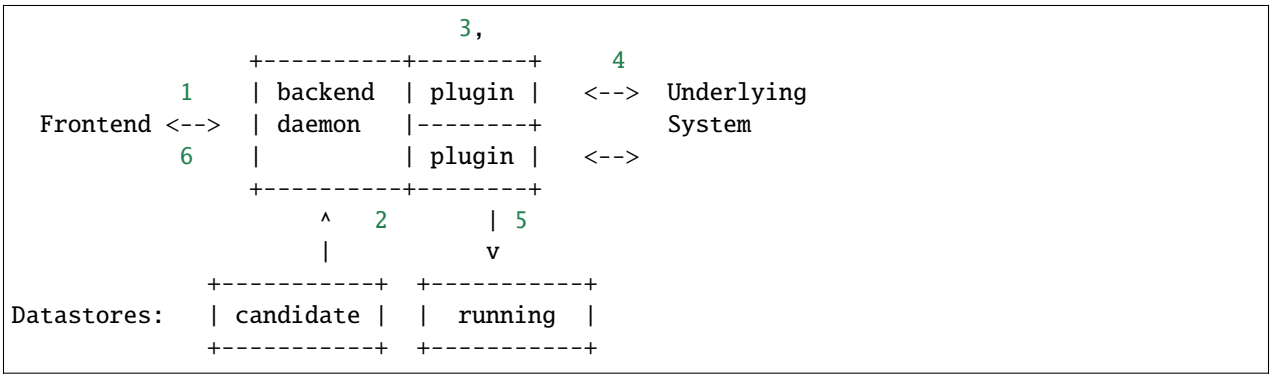
3. The backend modifies the *candidate* datastore. If there was no previous content it will look like the following after the edit:

```
<config>
  <hello xmlns="urn:example:hello">
    <world/>
  </hello>
</config>
```

4. The backend will reply with an OK:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply
```

14.3 14.3 Commit



After one, or several, edits, the user can commit the changes to running which triggers commit callbacks that will actually change the underlying system. Often, commits are made at once after every edit (such as RESTCONF operations). In that case, the edit described in the previous sections and commit are made in series by the client.

1. The client sends the commit message (frontend is not specified in this usecase):

```
<rpc username="olof">
  <commit/>
</rpc>
```

2. When the backend receives the commit message, it computes the differences between candidate and running datastores, creates a transaction data structure and initiates a transaction.
3. Each plugin in turn gets callbacks to validate the transaction. The plugins verifies that the proposed changes to the system is sound. If not, the commit fails.
4. Each plugin in turn gets callbacks to commit the transaction to the underlying system. In this step, the application-dependent API:s are used to push the changes made.
5. If all validation and callbacks succeed, running is replaced with current

6. An OK is returned to the user.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply
```

14.4 14.4 RESTCONF RPC

```
                              4   1
                        +----------+--------+    5
      2             3   | backend  | plugin |   <-->  Underlying
CURL <--> Restconf <--> | daemon   |--------+         System
      7   frontend  6   |          |
                        +----------+
```

A plugin can register an application-dependent RPC, and a client can then access it.

1. A plugin registers *example-rpc*:

```
rpc_callback_register(h, example_rpc, NULL, "urn:example:clixon", "example");
```

2. A user makes an RPC call, in this case RESTCONF:

```
curl -is -X POST -H "Content-Type: application/yang-data+json" -d '{"clixon-example:input
↪":{"x":0}}' http://localhost/restconf/operations/clixon-example:example
```

3. The restconf client receives the HTTP POST message (via a reverse proxy such as nginx) and translates the JSON to internal NETCONF:

```
<rpc username="none">
  <example xmlns="urn:example:clixon">
    <x>0</x>
  </example>
</rpc>
```

4. The backend receives the Netconf message and calls the registered callback *example_rpc()* in the plugin.
5. The plugin processes the rpc, for example by accessing state in the underlying system

6. The plugin returns a reply which is returned to the restonf client (for example):

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <x xmlns="urn:example:clixon">0</x>
  <y xmlns="urn:example:clixon">42</y>
</rpc-reply>
```

7. The restconf client translates the Netconf message to JSON and returns to the client (via a reverse proxy):

```
{
  "clixon-example:output":{
     "x":"0",
     "y":"42"
  }
}
```

CHAPTER

FIFTEEN

15 CLIENT API

Note: This section is not complete and outdated. A new Client API is developed as part of the Clixon controller.

Clixon's client API provides a way to communicate with the built-in XML datastore. This can be used to fetch or manipulate configurations handled by Clixon from any other application running on a host system. For example, a deamon running on a host system may need to read a configured value from the Clixon datastore.

Clixon integration normally uses dynamic plugins, but the client API shown here is an alternative.

Below is a minimal example application which connects to Clixon to get a value stored under "/table/parameter" in the XML store:

```
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>

#include <clixon/clixon_client.h>

int main(int argc, char **argv)
{
    clixon_handle h = NULL;
    clixon_client_handle ch = NULL;

    uint32_t n = 0;

    if ((h = clixon_client_init("/usr/local/etc/example.xml")) == NULL)
        return -1;

    if ((ch = clixon_client_connect(h, CLIXON_CLIENT_NETCONF)) == NULL)
        return -1;

    if (clixon_client_get_uint32(ch, &n, "urn:example:clixon", "/table/parameter[name='a
↪']/value") < 0)
        return -1;

    printf("Response: %n\n", u);

    clixon_client_disconnect(ch);
    clixon_client_terminate(h);
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Clixon data paths use full XPATHs:

```
/table/parameter[name='a']/value
```

One can make the same index access as well (eg *[0]*). This means that one can make direct indexed accesses as an alternative to looping.

15.1 15.1 Clixon and ConfD Examples

This describes how to create a minimal YANG specification and use it together with Clixon. First, a YANG model is added. Then, NETCONF messages are sent to Clixon which is thereafter accessed in the Clixon CLI.

This guide assumes that you have Clixon installed and running. Please refer to the respective for installation and initial configuration.

15.1.1 15.1.1 YANG model

The YANG model consists of a table with a list of parameters where each parameter have a name and a value:

```
module example {
    yang-version 1.1;
    namespace "urn:example:clixon";
    description
      "Tiny example to be used with Clixon and ConfD.
         ";
    revision 2021-01-26 {
      description "Added example.";
    }

    container example{
      list parameter{
          key name;
          leaf name{
              type string;
          }
          leaf value{
              type string;
          }
      }
    }
}
```

The YANG model is saved as *example.yang*.

15.1.2 15.1.2 Installating the YANG model

It is assumed that the example application shipped with Clixon is installed. If not it can be found in the source tree under "example/main".

The YANG file *example.yang* is copied to the folder where Clixon expects to find YANG models (usually */usr/local/share/clixon*):

```
$ sudo cp example.yang /usr/local/share/clixon/example.yang
```

The Clixons configuration should look something like:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_CONFIGFILE>/usr/local/etc/example.xml</CLICON_CONFIGFILE>
  <CLICON_FEATURE>ietf-netconf:startup</CLICON_FEATURE>
  <CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
  <CLICON_YANG_MODULE_MAIN>example</CLICON_YANG_MODULE_MAIN>
  <CLICON_CLI_MODE>example</CLICON_CLI_MODE>
  <CLICON_BACKEND_DIR>/usr/local/lib/example/backend</CLICON_BACKEND_DIR>
  <CLICON_NETCONF_DIR>/usr/local/lib/example/netconf</CLICON_NETCONF_DIR>
  <CLICON_RESTCONF_DIR>/usr/local/lib/example/restconf</CLICON_RESTCONF_DIR>
  <CLICON_CLI_DIR>/usr/local/lib/example/cli</CLICON_CLI_DIR>
  <CLICON_CLISPEC_DIR>/usr/local/lib/example/clispec</CLICON_CLISPEC_DIR>
  <CLICON_SOCK>/usr/local/var/example/example.sock</CLICON_SOCK>
  <CLICON_BACKEND_PIDFILE>/usr/local/var/example/example.pidfile</CLICON_BACKEND_PIDFILE>
  <CLICON_XMLDB_DIR>/usr/local/var/example</CLICON_XMLDB_DIR>
  <CLICON_CLI_LINESCROLLING>0</CLICON_CLI_LINESCROLLING>
  <CLICON_STARTUP_MODE>init</CLICON_STARTUP_MODE>
  <CLICON_NACM_MODE>disabled</CLICON_NACM_MODE>
  <CLICON_STREAM_DISCOVERY_RFC5277>true</CLICON_STREAM_DISCOVERY_RFC5277>
</clixon-config>
```

After this is done, the Clixon backend can be restarted, and the new model should be present.

15.1.3 15.1.3 Testing with NETCONF

The next step is to modify configuration values with NETCONF. A new *test* parameter is added with value *1234*.

In the example, NETCONF is running over SSH. The SSH configuration needs to contain the following line:

```
Subsystem netconf /usr/local/bin/clixon_netconf -f /usr/local/etc/example.xml
```

The following NETCONF operation is used:

```
<?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:1.1</capability>
  </capabilities>
</hello>
]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <edit-config>
    <target>
```

(continues on next page)

(continued from previous page)

```
      <running/>
    </target>
    <config>
      <table xmlns="urn:example:clixon">
        <parameter>
          <name>test</name>
          <value>1234</value>
      </parameter>
      </table>
    </config>
  </edit-config>
</rpc>
]]>]]>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">
  <commit/>
</rpc>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">
  <close-session/>
</rpc>
]]>]]>
```

The XML is saved as "example.xml" and use the following commands to test it:

```
$ ssh 192.168.1.56 -s netconf < example.xml
```

If everything went fine, a reply is returned saying OK:

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

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">
  <ok/>
</rpc-reply>
]]>]]>
```

Finally, the config can be viewed from the CLI:

```
root@debian10-clixon /> show configuration
example {
    parameter {
        name test;
        value 1234;
    }
}
```

CHAPTER

SIXTEEN

16 XML AND PATHS

This section describes XML trees and how to navigate in trees using paths.

Clixon represents its internal data in an in "in-memory" tree representation. In the C API, this data structure is called `cxobj` (Clixon XML object) and is used to represent config and state data. Typically, a cxobj is parsed from or printed to XML or JSON, but is really a generic representation of a tree.

16.1 16.1 Paths

Clixon uses paths to navigate in XML trees. Clixon uses the following three methods:

- *XML Path Language* defined in XPath 1.0 as part of XML and used in NETCONF.
- *Instance-identifier* defined in RFC 7950: The YANG 1.1 Data Modeling Language, a subset of XPath and used in NACM,
- *Api-path* defined and used in RFC 8040: RESTCONF Protocol

16.1.1 16.1.1 XPath

Example of XPath in a NETCONF *get-config* RPC using the XPath capability:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <get-config>
      <source>
         <candidate/>
      </source>
      <filter type="xpath" select="/interfaces/interface[name='eth0']/description" />
   </get-config>
</rpc>
```

XPath is a powerful language for addressing parts of an XML document, including types and expressions. The following is a valid but complex XPath:

```
/assembly[name="robot_4"]//shape/name[contains(text(),'bolt')]/surface/roughness
```

Clixon uses XPaths extensively due to their expressive power. However, it is recommended to use instance-identifiers instead if you want optimized access.

16.1.2 16.1.2 Namespaces in XPaths

XPath uses XML names, requiring an *XML namespace context* using the *xmlns* attribute to bind namespaces and prefixes. An XML namespace context can specify both:

- A default namespace for unprefixed names (*/x/*), defined by for example: *xmlns="urn:example:default"*.
- An explicit namespace for prefixed names prefix (*/ex:x/*), defined by for example: *xmlns:ex="urn:example:example"*.

Further, XML prefixes are *not inherited*, each symbol must be prefixed with a prefix or default. That is, */ex:x/y* is not the same as */ex:x/ex:y*, unless *ex* is also default.

Example: Assume an XML namespace context:

```
<a xmlns="urn:example:default" xmlns:ex="urn:example:example">
```

with an associated XPath:

```
/x/ex:y/ex:z[ex:i='w']`,
```

then symbol *x* belongs to "urn:example:default" and symbols *y*, *z* and *i* belong to "urn:example:example".

16.1.3 16.1.3 Instance-identifier

Instance-id:s are defined in YANG for some minor usage but appears in for example NACM and provides a useful subset of XPath. The subset is as follows (see Section 9.13 in YANG 1.1):

- Child paths using slashes: `/ex:system/ex:services`
- List entries for one or several keys: `/ex:system[ex:ip='192.0.2.1'][ex:port='80']`
- Leaf-list entries for one key: `/ex:system/ex:cipher[.='blowfish-cbc']`
- Position in lists: `/ex:stats/ex:port[3]`

Example of instance-id in NACM:

```
<path xmlns:acme="http://example.com/ns/itf">
      /acme:interfaces/acme:interface[acme:name='dummy']
</path>
```

Namespaces in instance-identifiers are the same as in XPaths.

16.1.4 16.1.4 Api-path

RESTCONF defines api-paths as a YANG-based path language. Keys are implicit which make path expressions more concise, but they are also less powerful

Example of Api-path in a restconf GET request:

```
curl -s -X GET http://localhost/restconf/data/ietf-interfaces:interfaces/interface=eth0/
↪description
```

Clixon uses Api-paths internally in some cases when accessing xml keys, but more commonly translates Api-paths to XPaths.

Api-path is in comparison to XPaths limited to pure path expressions such as, for example:

```
a/b=3,4/c
```

which corresponds to the XPath: *a[i=3][j=4]/c*. Note that you cannot express any other index variables than the YANG list keys.

16.1.5 16.1.5 Namespaces in Api-paths

In contrast to XPath, Api-path namespaces are defined implicitly by a YANG context using *module-names* as prefixes. The namespace is defined in the Yang module by the *namespace* keyword. Api-paths must have a Yang definition whereas XPaths can be completely defined in XML.

A prefix/module-name is *inherited*, such that a child inherits the prefix of a parent, and there are no defaults. For example, */moda:x/y* is the same as */moda:a/moda:y*.

Further, an Api-path uses a shorthand for defining list indexes. For example, */modx:z=w* denotes the element in a list of *z*:s whose key is the value *w*. This assumes that *z* is a Yang list (or leaf-list) and the index value is known.

Example: Assume two YANG modules *moda* and *modx* with namespaces "urn:example:default" and "urn:example:example" respectively, with the following Api-path (equivalent to the XPath example above):

```
/moda:x/modx:y/z=w
```

where, as above, *x* belongs to "urn:example:default" and *y*, and *z* belong to "urn:example:example".

16.2 16.2 XML trees

XML objects are typed as follows XML 1.0:

- Element: non-terminal node with child nodes
- Attribute: name value pair
- Body: text between tags

Elements and attributes have names. An element has a set of children while body and attribute have values.

For example, the following XML tree:

```
<y xmlns="urn:example:a">
   <x>
      <k>a</k>
   </x>
</y>
```

can have an internal tree representation as follows (`e` represents XML element, `b` body and `a` attribute:

```
y:e ---------> xmlns:a (value:"urn:example:a")
     \
       +-----> x:e ---------> k:e ---------> :b (value:"a")
```

16.2.1 16.2.1 Yang binding

Typically, XML elements are associated with a YANG data node specification(`yang_stmt`). A YANG data node is either a container, leaf, leaf-list, list, or anydata.

A YANG bound XML node have some constraints with respect to children as follows:

- All elements may have attributes
- A leaf or leaf-list has at most one body child.
- A leaf or leaf-list have no elements children
- A container or list have no body children
- A container or list may have one or many element children
- An anydata node may have both elements and one body child(?>

The XML example given earlier could have the following YANG specification:

```
module mod_a{
  prefix a;
  namespace "urn:example:a";
  container y {
    list x{
      key k;
      leaf k{
        type string;
      }
    }
  }
}
```

Annotating the tree representation with YANG specification, could yield the following YANG bound tree:

```
container y
y:e ---------> xmlns:a (value:"urn:example:a")
   \
    \          list x         leaf k
      +------> x:e ---------> k:e ---------> :b (value:"a")
```

16.2.2 16.2.2 Sorted tree

Once an XML tree is bound to YANG, it can be sorted.

The tree is sorted using a "multi-layered" approach:

1. XML type: attributes come before elements and bodies.
2. Yang nodename: XML nodes with same nodename are adjacent and follow the order they are given in the Yang specification. If XML nodes belong to different modules, they follow the order they were loaded into the system.
3. Sorting lists and leaf-lists. There are two variants:

a) Ordered-by-system: This is the default. Elements are sorted according to key value. Key value comparison is typed: if the key type is string, strcmp is used, if the key value is an integer, integer <>= is used, etc.

b) Ordered-by-user: the list items follow the ordered they were entered, regardless of list keys. Ordered-by-user is not recommended in clixon since the optimized searching algorithms uses sorted lists.

Extending the example above slightly with a new list `x2` as follows:

```
list x2{
  key k2;
  leaf k2{
    type int32;
  }
}
```

could give the following sorted XML tree:

```
<y xmlns="urn:example:a">
   <x>
      <k>a</k>
   </x>
    <x>
      <k>b</k>
   </x>
   <x2>
      <k2>9</k2>
   </x2>
   <x2>
      <k2>100</k2>
   </x2>
</y>
```

Note that among `y`:s children, the attribute is the first (layer 1), then follows the group of `x` elements and the group of `x2` elements as they are given in the YANG specification (layer 2). Finally, the lists are internally sorted according to key values.

Note: Sorting is necessary to achieve fast searching as described in Section *Searching in XML*.

16.3 16.3 Creating XML

The creation of and XML tree goes thorough three steps:

1. Syntactic creation. This is done either via parsing or via manual API calls.
2. Bind to YANG. Assure that the XML tree complies to a YANG specification.
3. Semantic validation. Ensuring that the XML tree complies to the backend validation rules.

Steps 2 and 3 are optional.

16.3.1 16.3.1 Creating XML from a string

A simple way to create an cxobj is to parse it from a string:

```
cxobj *xt = NULL;
if ((ret = clixon_xml_parse_string("<y xmlns='urn:example:a'><x><k1>a</k2></x></y>",
                    YB_MODULE, yspec, &xt, NULL)) < 0)
   err;
if (ret == 0)
   err; /* yang error */
```

where

- `YB_MODULE` is the default Yang binding mode, see *Binding YANG to XML*.
- `xt` is a top-level cxobj containing the XML tree.
- `yspec` is the top-level yang spec obtained by e.g., `clicon_dbspec_yang(h)`

If printed with for example: `xml_print(stdout, xt)` the tree looks as follows:

```
<top>
   <y xmlns="urn:example:a">
     <x>
       <k1>a</k1>
     </x>
   </y>
</top>
```

Note that a top-level node (`top`) is always created to encapsulate all trees parsed and that the default namespace in this example is "urn:example:a".

The XML parse API has several other functions, including:

- `clixon_xml_parse_file()` Parse a file containing XML
- `clixon_xml_parse_va()` Parse a string using variable argument strings

16.3.2 16.3.2 Creating JSON from a string

You can create an XML tree from JSON as well:

```
cxobj *xt = NUL;L
cxobj *xerr = NULL;

if ((ret = clixon_json_parse_string("{\"mod_a:y\":{\"x\":{\"k1\":\"a\"}}}",
                YB_MODULE, yspec, &xt, NULL)) < 0)
   err;
```

yielding the same xt tree as in *Creating XML from a string*.

In JSON, namespace prefixes use YANG module names, making the JSON format dependent on a correct YANG binding.

The JSON parse API also includes:

- `clixon_json_parse_file()` Parse a file containing JSON

16.3.3 16.3.3 Creating XML programmatically

You may also manually create a tree by `xml_new()`, `xml_new_body()`, `xml_addsub()`, `xml_merge()` and other functions. Instead of parsing a string, a tree is built manually. This may be more efficient but more work to program.

The following example creates the same XML tree as in the above examples using API calls:

```
cxobj *xt, *xy, *x, *xa;
if ((xt = xml_new("top", NULL, CX_ELMNT)) == NULL)
   goto done;
if ((xy = xml_new("y", xt, CX_ELMNT)) == NULL)
   goto done;
if ((xa = xml_new("xmlns", y, CX_ATTR)) == NULL)
   goto done;
if (xml_value_set(xa, "urn:example:a") < 0)
   goto done;
if ((x = xml_new("xy", xt, CX_ELMNT)) == NULL)
   goto done;
if (xml_new_body("k1", x, "a") == NULL)
   goto done;
```

Note: If you create the XML tree manually, you may have to explicitly call a yang bind function.

16.3.4 16.3.4 Binding YANG to XML

A further step is to ensure that the XML tree complies to a YANG specification. This is an optional step since you can handle XML without YANG, but often necessary in Clixon, since some functions require YANG bindings to be performed correctly. This includes sort, validate, merge and insert functions, for example.

Yang binding may be done already in the XML parsing phase, and is mandatory for JSON parsing. If XML is manually created, you need to explicitly call the Yang binding functions.

For the XML in the example above, the YANG module could look something like:

```
module mod_a{
  prefix a;
  namespace "urn:example:a";
  container y {
    list x{
      key k1;
      leaf k1{
        type string;
      }
    }
  }
}
```

Binding is made with the `xml_bind_yang()` API. The bind API can be done in some different ways as follows:

- `YB_MODULE` Search for matching yang binding among top-level symbols of Yang modules. This is default.
- `YB_PARENT` Assume yang binding of existing parent and match its children by name
- `YB_NONE` Do not bind

In the example above, the binding is `YB_MODULE` since the top-level symbol `x` is a top-level symbol of a module.

But assume instead that the string `<k1 xmlns="urn:example:a">a</k1>` is parsed or created manually. You can determine which module it belongs to from the namespace, but there may be many `k1` symbols in that module, you do not know if the "leaf" one in the Yang spec above is the correct one.

The following is an example of how to bind yang to an XML tree `xt`:

```
cxobj *xt;
cxobj *xerr = NULL;
/* create xt as example above */
if ((ret = xml_bind_yang(h, xt, YB_MODULE, yspec, NULL)) < 0)
   goto done;   /* fatal error */
if (ret == 0)
   goto noyang; /* yang binding error */
```

The return values from the bind API are same as parsing, as follows:

- 1 OK yang assignment made
- `0` Partial or no yang assignment made (at least one failed) and xerr set
- -1 Error

As an example of *YB_PARENT* Yang binding, the `k1` subtree is inserted under an existing XML tree which has already been bound to YANG. Such as an XML tree with the `x` symbol.

16.3.5 16.3.5 Config data

To create a copy of configuration data, a user retrieve a copy from the datastore to get a cxobj handle. This tree is fully bound, sorted and defaults set. Read-only operations may then be done on the in-memory tree.

The following example code gets a copy of the whole *running* datastore to cxobj `xt`:

```
cxobj *xt = NULL;
if (xmldb_get(h, "running", NULL, NULL, &xt) < 0)
   err;
```

Note: In the case of config data, in-memory trees are read-only *caches* of the datastore and can normally not be written back to the datastore. Changes to the config datastore should be made via the backend netconf API, eg using `edit-config`.

16.4 16.4 Modifying XML

Once an XML tree has been created and bound to YANG, it can be modified in several ways.

16.4.1 16.4.1 Merging

If you have two trees, you can merge them with `xml_merge` as follows:

```
if ((ret = xml_merge(xt, x2, yspec, &reason)) < 0)
  err;
if (ret == 0)
  err; /* yang failure */
```

where both `xt` and `x2` are root XML trees (directly under a module) and fully YANG bound. For example, if `x2` is:

```
<top>
   <y xmlns="urn:example:a">
     <x>
       <k1>z</k1>
     </x>
   </y>
</top>
```

the result tree `xt` after merge is:

```
<top>
   <y xmlns="urn:example:a">
     <x>
       <k1>a</k1>
     </x>
     <x>
       <k1>z</k1>
     </x>
   </y>
</top>
```

Note that the result tree is sorted and YANG bound as well.

16.4.2 16.4.2 Inserting

Inserting a subtree can be made in several ways. The most straightforward is using parsing and the `YB_PARENT` YANG binding:

```
cxobj *xy;
xy = xpath_first(xt, NULL, "%s", "y");
if ((ret = clixon_xml_parse_string("<x><k1>z</k2></x>", YB_PARENT, yspec, &xy, NULL)) <␣
↪0)
if (ret == 0)
   err; /* yang error */
```

with the same result as in tree merge.

Note that `xy` in this example points at the `y` node and is where the new tree is pasted. Neither tree need to be a root tree.

Another way to insert a subtree is to use `xml_insert`:

```
if (xml_insert(xy, xi, INS_LAST, NULL, NULL) < 0)
   err;
```

where both `xy` and `xi` are YANG bound trees. It is possible to specify where the new child is inserted (last in the example), but this only applies if `ordered-by user` is specified in YANG. Otherwise, the system will order the insertion of the subtree automatically.

16.4.3 16.4.3 Removing

A subtree can be permanently removed, or just pruned in order to insert it somewhere else. and graft subtrees.

Permanently deleting a (sub)tree `x` and remove or from its parent is done as follows:

```
xml_purge(x);
```

Removing a subtree `x` from its parent is done as follows:

```
xml_rm(x);
```

or alternatively remove child number `i` from parent `xp`:

```
xml_child_rm(xp, i);
```

In both these cases, the child `x` can be used as a stand-alone tree, or being inserted under another parent.

16.4.4 16.4.4 Copying

An XML tree `x0` can be copied as follows:

```
cxobj *x1;
x1 = xml_new("new", NULL, xml_type(x0));
if (xml_copy(x0, x1) < 0)
   err;
```

Alternatively, a tree can be duplicated as follows:

```
x1 = xml_dup(x0);
```

In these cases, the new object `x1` can be use as a separate tree for insertion, for example.

16.5 16.5 Searching in XML

Clixon search indexes are either *implicitly* created from the YANG specification, or *explicitly* created using the API.

From YANG it is only `list` and `leaf-list` that are candidates for optimized lookup, direct `leaf` and `container` lookup is fast either way.

Binary search is used by search indexes and works by ordering list items alphabetically (or numerically), and then dividing the search interval in two equal parts depending on if the requested item is larger than, or less than, the middle of the interval.

Binary search complexity is *O(log N)*, whereas linear search is is *O(n)*. For example, a search in a vector of one million children will take up to *20* lookups, whereas linear search takes up to *1.000.000* lookups.

Therefore, if you have a large number of children and you need to make searches, it is important that you use indexes, either implicit, or explicit.

16.5.1 16.5.1 Auto-generated indexes

Auto-generated (or implicit) YANG-based search indexes are based on `list` and `leaf-lists`. For any list with keys `k1,...kn`, a set of indexes are created and an optimized search can be made using the keys in the order they are defined.

For example, assume the following YANG (this YANG is reused in later examples):

```
module mod_a{
  prefix a;
  namespace "urn:example:a";
  import clixon-config {
    prefix "cc";
  }
  list x{
    key "k1 k2";
    leaf k1{
      type string;
    }
    leaf k2{
      type string;
    }
    leaf-list y{
      type string;
    }
    leaf z{
      type string;
    }
    leaf w{
      type string;
      cc:search_index;
    }
    ...
```

Assume also an example XML tree as follows:

```
<top xmlns="urn:example:a">
  <x>
    <k1>a</k1>
    <k2>a</k2>
    <y>cc</y>
    <y>dd</y>
    <z>ee</z>
    <w>ee</w>
  </x>
  <x>
    <k1>a</k1>
    <k2>b</k2>
    <y>cc</y>
    <y>dd</y>
    <z>ff</z>
    <w>ff</w>
  </x>
  <x>
    <k1>b</k1>
```

(continues on next page)

(continued from previous page)

```
    ...
</top>
```

Then there will be two implicit search indexes created for all XML nodes `x` so that they can be accessed with *O(log N)* with e.g.:

- XPath or Instance-id: `x[k1="a"][k2="b"]`.
- Api-path: `x=a,b`.

If other search variables are used, such as: `x[z="ff"]` the time complexity will be *O(n)* since there is no explicit index for `z`. The same applies to using key variables in another order than they appear in the YANG specification, eg: `x[k2="b"][k1="a"]`.

A search index is also generated for leaf-lists, using `x` as the base node, the following searches are optimized:

- XPath or Instance-id: `y[.="bb"]`.
- Api-path: `y=bb`.

In the following cases, implicit indexes are *not* created:

- No YANG definition of the XML children exists. There are several use-cases. For example that YANG is not used or the tree is part of YANG *ANYXML*.
- The list represents *state* data
- The list is *ordered-by user* instead of the default YANG *ordered-by system*.

16.5.2 16.5.2 Explicit indexes

In those cases where implicit YANG indexes cannot be used, indexes can be explicitly declared for fast access. Clixon uses a YANG extension to declare such indexes: *search_index* as shown in the example above for leaf `w`:

```
leaf w{
  type string;
  cc:search_index;
}
```

In this example, `w` can be used as a search index with *O(log N)* in the search API.

The corresponding direct API call is: `yang_list_index_add()`

16.5.3 16.5.3 Direct children

The basic C API for searching direct children of a cxobj is the `clixon_xml_find_index()` API.

An example call is as follows:

```
clixon_xvec *xv = NULL;
cvec    *cvk = NULL;

if ((xv = clixon_xvec_new()) == NULL)
   goto done;
/* Populate cvk with key/values eg k1=a k2:b */
if (clixon_xml_find_index(xp, yp, namespace, name, cvk, xv) < 0)
   err;
```

(continues on next page)

(continued from previous page)

```
/* Loop over found children*/
for (i = 0; i < clixon_xvec_len(xv); i++) {
    x = clixon_xvec_i(xvec, i);
    ...
}
if (xv)
   clixon_xvec_free(xv);
```

where

xp	is an XML parent
yp	is the YANG specification of xp
name	is the name of the wanted children
cvk	is a vector of index name and value pairs
xvec	is a result vector of XML nodes.

For example, using the previous XML tree and if `name=x` and `cvk` contains the single pair: `k1=a`, then `xvec` will contain both `x` entries after calling the function:

```
0: <x><k1>a</k1><k2>a</k2><y>cc</y><y>dd</y><z>foo</a></x>
1: <x><k1>a</k1><k2>b</k2><y>cc</y><y>dd</y><z>bar</a></x>
```

and the search was done using *O(logN)*.

16.5.4 16.5.4 Using paths in XML

If deeper searches are needed, i.e., not just to direct children, Clixon *paths* can be used to make a search request. There are three path variants, each with its own pros and cons:

- XPath is most expressive, but only supports *O(logN)* search for YANG *list* entries (not leaf-lists), and adds overhead in terms of memory and cycles.
- Api-path is least expressive since it can only express YANG *list* and *leaf-list* key search.
- Instance-identifier can express all optimized searches as well as non-key searches. This is the recommended option.

Assume the same YANG as in the previous example, a path to find `y` entries with a specific value could be:

- XPath or instance-id: `/a:x[a:k1="a"][a:k2="b"]/a:y[.="bb"]`
- Api-path: `/mod_a:x=a,b/y=bb`

which results in the following result:

```
0: <y>bb</y>
```

An example call using instance-id:s is as follows:

```
cxobj **vec = NULL;
size_t  len = 0;
if (clixon_xml_find_instance_id(xt, yt, &vec, &len,
       "/a:x[a:k1=\"a\"][k2=\"b\"]/a:y[.=\"bb\""") < 0)
   goto err;
```

(continues on next page)

(continued from previous page)

```
for (i=0; i<len; i++){
   x = vec[i];
      ...
}
```

The example shows the usage of auto-generated key indexes which makes this work in *O(logN)*, with the same exception rules as for direct children state in *Auto-generated indexes*.

An example call using api-path:s instead is as follows:

```
cxobj **vec = NULL;
size_t  len = 0;
if (clixon_xml_find_api_path(xt, yt, &vec, &len,
       "/mod_a:x=a,b/y=bb") < 0)
   goto err;
for (i=0; i<len; i++){
   x = vec[i];
      ...
}
```

The corresponding API for XPath is `xpath_vec()`.

16.5.5 16.5.5 Multiple keys

Optimized *O(logN)* lookup works with multiple key YANG *lists* but not for explicit indexes. Further, less significant keys can be omitted which may result multiple result nodes.

For example, the following lookups can be made using *O(logN)* using implicit indexes:

```
x[k1="a"][k2="b"]/y[.="cc"]
x[k1="a"]/y[.="cc"]
x[k1="a"][k2="b"]
```

The following lookups are made with *O(N)*:

```
x[k2="b"][k1="a"]
x[k1="a"][z="foo"]
```

16.6 16.6 Internal representation

A cxobj has several components, which are all accessible via the API. For example:

name	Name of node
prefix	Optional prefix denoting a localname according to XML namespaces
type	A node is either an element, attribute or body text
value	Attributes and bodies may have values.
children	Elements may have a set of XML children
spec	A pointer to a YANG specification of this XML node

The most basic way to traverse an cxobj tree is to linearly iterate over all children from a parent element node.

```
cxobj *x = NULL;
while ((x = xml_child_each(xt, x, CX_ELMNT)) != NULL) {
  ...
}
```

where CX_ELMNT selects element children (no attributes or body text).

However, it is recommended to use the *Searching in XML* for more efficient searching.

16.7 16.7 Character encoding

Clixon implements encoding of character data as defined in XML 1.0, Section 2.4.

It can be illustrated by some examples. Assume a data-field "value" of type "string" including some special characters (wrt XML): "<description/>". This string can be input using NETCONF or RESTCONF using XML and JSON as follows:

1. Restconf POST using JSON, eg: `{"value": "<description/>"}`
2. Restconf POST using XML regular x3 encoding, eg: `<value>&lt;description/&gt;</value>`
3. Restconf POST using XML and CDATA: `<value><![CDATA[<description/>]]></value>`
4. Netconf edit-config using XML regular encoding: `<value>&lt;description/&gt;</value>`
5. Netconf edit-config using XML CDATA: `<value><![CDATA[<description/>]]></value>`

The input is received by the backend where the value is stored in the backend as follows:

1. `<value>&lt;description/&gt;</value>`
2. `<value>&lt;description/&gt;</value>`
3. `<value><![CDATA[<description/>]]></value>`
4. `<value>&lt;description/&gt;</value>`
5. `<value><![CDATA[<description/>]]></value>`

Note that in most cases, the data is just propagated from input to datastore, except in the JSON to XML translation(case 1).

For output assuming the value above, there are two data formats to consider in the datastore above: 1) with regular x3 encoding and 2) using CDATA.

There are the following cases:

1. Restconf GET datastore entry 1 using JSON: `"{"value": "<description/>"}`
2. Restconf GET datastore entry 2 using JSON: `"{"value": "<![CDATA[<description/>]]>"}` (recently changed)
3. Restconf GET datastore entry 1 using XML: `<value>&lt;description/&gt;</value>`
4. Restconf GET datastore entry 2 using XML: `<value><![CDATA[<description/>]]></value>`
5. Netconf get-config datastore entry 1: `<value>&lt;description/&gt;</value>`
6. Netconf get-config datastore entry 2: `<value><![CDATA[<description/>]]></value>`

Internally, data is saved in cleartext which is encoded when translated to XML. CDATA encoding is an exception where it is stored internally as well.

CHAPTER

SEVENTEEN

17 PAGINATION

Pagination and scrolling is the technique of showing large amount of data in smaller chunks. Pagination was introduced in Clixon 5.3 and updated in 5.4. Expect further development and changes.

17.1 17.1 Overview

The pagination solution is based on the following drafts:

- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-00.html
- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-nc-00.html
- https://www.ietf.org/archive/id/draft-ietf-netconf-list-pagination-rc-00.html

The drafts define two YANGs:

- ietf-list-pagination@2022-07-24.yang
- ietf-list-pagination-nc@2022-07-24.yang

The pagination in Clixon is currently restricted to the *offset* and *limit* attributes. For example, the following requests a list of 120 items in chunks of 20:

```
get offset=0 limit=20
get offset=20 limit=20
...
get offset=100 limit=20
```

This is referred to as *stateless* pagination, since the state may change between "get" calls. For paging of a consistent snapshot view, consider *locked* pagination.

Clixon pagination encompasses several aspects:

1. Locked pagination
2. NETCONF/RESTCONF protocol extensions
3. CLI scrolling
4. Plugin state API

17.2 17.2 Locked pagination

Using NETCONF, one can lock the datastore during a session to ensure that the data is unchanged, such as:

```
lock-db
get offset=0 limit=20
get offset=20 limit=20
...
get offset=100 limit=20
unlock-db
```

The use of locks guarantees a consistent view (snapshot) of config data, but risks undefinite blocking in a CLI pagination situation for example.

For state pagination data, database locks are not useful. Instead clixon extends the database lock mechanism with a specific *state-paginate-lock* lock that can be used by the state paginate callback developer. It works the same way as a database lock, but there is no config database associated with it, it is specially made for paginated state data only.

17.3 17.3 Pagination protocol

In RESTCONF a pagination request looks as follows:

```
GET /localhost/restconf/data/example-social:members/uint8-numbers?offset=20&limit=10␣
↪HTTP/1.1
Host: example.com
Accept: application/yang-collection+json
```

In NETCONF a similar request is:

```
<rpc>
   <get>
      <filter type="xpath" select="/es:members/es:uint8-numbers" xmlns:es="http://
↪example.com/ns/example-social"/>
      <list-pagination xmlns="urn:ietf:params:xml:ns:yang:ietf-list-pagination-nc">
         <offset>20</offset>
         <limit>10</limit>
      </list-pagination>
   </get>
</rpc>
```

In return, Clixon returns a reply with the requested number of entries (in NETCONF):

```
<rpc-reply>
   <data>
      <members xmlns="http://example.com/ns/example-social">
         <member>
            <member-id>alice</member-id>
            <privacy-settings>
               <post-visibility>public</post-visibility>
            </privacy-settings>
            <favorites>
               <uint8-numbers>20</uint8-numbers>
```

(continues on next page)

(continued from previous page)

```
                <uint8-numbers>21</uint8-numbers>
                ...
                <uint8-numbers>29</uint8-numbers>
              </favorites>
            </member>
          </members>
        </data>
</rpc-reply>
```

17.4 17.4 CLI scrolling

CLIgen has a scrolling mechanism that can be integrated with pagination. For example, showing the list from the example above:

```
clixon_cli
cli> show state /es:members/es:member[es:member-id=\'bob\']/es:favorites/es:uint64-
↪numbers cli
uint64-numbers 0
uint64-numbers 1
...
uint64-numbers 9
--More--
```

17.4.1 17.4.1 CLI callbacks

CLI scrolling is implemented by the *cligen_output* function similar to *printf* in syntax. By using cligen_output for all output, CLIgen ensures a scrolling mechanism.

Clixon includes an example CLI callback function that combines the scrolling mechanism of the CLI with NETCONF pagination called *cli_pagination* with the following arguments:

- *xpath* : XPath of a leaf-list or list
- *prefix* : Prefix used in XPath (only one can be specified)
- *namespace* : Namespace associated with prefix
- *format* : one of xml, text, json, or cli
- *limit* : Number of lines of the pagination window

In the main example, cli_pagination is called as follows:

```
show state <xpath:string> cli, cli_pagination("", "es", "http://example.com/ns/example-
↪social", "cli", "10");
```

An application can use the *cli_pagination* callback, or create a tailor-made CLI callback based on the example callback.

17.5 17.5 Backend pagination API

While pagination of config data is built-in, state data needs backend plugin callbacks. There is a special state pagination callback API where a callback is bound to an xpath, and is called when a pagination request is made on an xpath.

Such a callback is registered with an XPath and a callback as follows:

```
clixon_pagination_cb_register(h, mycallback, "/myxpath", myarg);
```

where the callback has the following signature:

```
int
mycallback(void            *h,
           char            *xpath,
           pagination_data  pd,
           void            *arg)
```

The pd parameter has the following accessor functions:

```
uint32_t pagination_offset(pagination_data pd)
uint32_t pagination_limit(pagination_data pd)
int      pagination_locked(pagination_data pd)
cxobj*   pagination_xstate(pagination_data pd)
```

Essentially, the state callback requests state data for list/leaf-list *xpath* in the interval *[offset...offset+limit]*.

If *locked* is true, the plugin can cache the state data, return further requests from the same cache until the lock on the "runníng" database is released, thus forming an (implicit) transaction. For this, the ca_lockdb callback can be used as an end to the transaction of `state-paginate-lock`. Note that there is not explicit "start transaction", the first locked pagination request acts as one.

See a detailed example in the main example.

CHAPTER

EIGHTEEN

18 UPGRADE

When Clixon starts, the backend reads a startup configuration (see *startup modes*) parses it, checks for upgrades, and validates the configuration. It can also add extra XML, outside of the normal startup. On errors, it can enter a failsafe mode.

Clixon has three upgrade methods:

- General-purpose datastore upgrade.
- Module-specific manual upgrade
- Automatic upgrade (experimental)

18.1 18.1 General-purpose

A plugin registers a *ca_datastore_upgrade* function which gets called once on startup. This upgrade should be seen as a generic wrapper function to basic repair or upgrade of existing datastores. The module-specific upgrade callbacks are more fine-grained.

The general-purpose upgrade callback is usable if module-state is not available, or actions such as the following need to be done in the whole datastore:

- Remove or rename nodes
- Rename namespaces
- Add nodes

A recommended method, as shown in the example, is to make a pattern matching using XPath and then perform actions on the resulting nodes.

Example:

```
static clixon_plugin_api api = {
   ...
   .ca_datastore_upgrade=example_upgrade
};

/*! General-purpose datastore upgrade callback called once on startup
 * @param[in] h    Clicon handle
 * @param[in] db   Name of datastore, eg "running", "startup" or "tmp"
 * @param[in] xt   XML tree. Upgrade this "in place"
 * @param[in] msd  Info on datastore module-state, if any
 */
int
```

(continues on next page)

(continued from previous page)

```
example_upgrade(clixon_handle h,
                char         *db,
                cxobj        *xt,
                modstate_diff_t *msd)
{
    cxobj   **xvec = NULL;   /* vector of result nodes */
    size_t    xlen;
    cvec     *nsc = NULL;    /* Canonical namespace */
    int       i;

    /* Skip other than startup datastore */
    if (strcmp(db, "startup") != 0)
       return 0;
    /* Skip if there is proper module-state in datastore */
    if (msd->md_status)
       return 0;
    /* Get canonical namespaces for using "normalized" prefixes */
    if (xml_nsctx_yangspec(yspec, &nsc) < 0)
       err;
    /* Get all xml nodes matching path */
    if (xpath_vec(xt, nsc, "/a:x/a:y", &xvec, &xlen) < 0)
       err;
    /* Iterate through nodes and remove them */
    for (i=0; i<xlen; i++){
       if (xml_purge(xvec[i]) < 0)
          err;
    }
}
```

The example above first checks whether it is the *startup* datastore and that it does not contain module-state. It then matches all nodes matching the XPath */a:x/a:y* using canonical prefixes, which are then deleted.

18.2 18.2 Module-specific upgrade

Module-specific upgrade is only available if module-state is enabled, see *Module library support*.

If the module-state of the startup configuration does not match the module-state of the backend daemon, a set of module-specific upgrade callbacks are made. This allows a user to program upgrade funtions in the backend plugins to automatically upgrade the XML to the current version.

A user registers upgrade callbacks per YANG module. A user can register many callbacks, or choose wildcards. When an upgrade occurs, the callbacks will be called if they match the module and the modules have changed.

A module has changed if one of the following is true: - A module present in the startup is no longer present in the system (DEL) - A module in the system is not present in the startup (ADD) - A module present in both the startup and the system has a different revision date (CHANGE)

18.2.1 18.2.1 Registering a callback

A user registers upgrade callbacks in the backend *clixon_plugin_init()* function. The signature of upgrade callback is as follows:

```
upgrade_callback_register(h, cb, ns, arg);
```

where:

- *h* is the Clicon handle,
- *cb* is the name of the callback function,
- *ns* defines the namespace of a Yang module. NULL denotes all modules.
- *arg* is a user defined argument which can be passed to the callback.

One example of registering an upgrade of an interface module:

```
upgrade_callback_register(h, upgrade_interfaces, "urn:example:interfaces", NULL);
```

18.2.2 18.2.2 Upgrade callback

When Clixon loads a startup datastore with outdated modules, the matching upgrade callbacks will be called.

The signature of an upgrade callback is as follows:

```
int upgrade_interfaces(h, xt, ns, op, from, to, arg, cbret)
```

where:

- *xt* is the XML tree to be upgraded
- *ns* is the namespace of the YANG module.
- *op* is a flag indicating upgrading operation, one of: `XML_FLAG_ADD`, `XML_FLAG_DEL`, `XML_FLAG_CHANGE`. Note that this applies to per-module: whether a *module* has been added, deleted or changed.
- *from* is the revision date in the startup file of the module. It is zero if the operation is `ADD`
- *to* is the revision date of the YANG module in the system. It is zero if the operation is `DEL`

If no action is made by the upgrade callback, and thus the XML is not upgraded, the next step is XML/Yang validation.

An out-dated XML may still pass validation and the system will go up in normal state.

However, if the validation fails, the backend will try to enter the failsafe mode so that the user may perform manual upgrading of the configuration.

18.2.3 18.2.3 Example upgrade

The Clixon main example shows code for upgrading of an interface module. The example is inspired by the ietf-interfaces module that made a subset of the upgrades shown in the examples.

The code is split in two steps. The *upgrade_2014_to_2016* callback does the following transforms:

- Move `/if:interfaces-state/if:interface/if:admin-status` to `/if:interfaces/if:interface/`
- Move `/if:interfaces-state/if:interface/if:statistics` to `if:interfaces/if:interface/`
- Rename `/interfaces/interface/description` to `/interfaces/interface/descr`

The *upgrade_2016_to_2018* callback does the following transforms:

- Delete `/if:interfaces-state`
- Wrap `/interfaces/interface/descr` to `/interfaces/interface/docs/descr`
- Change type `/interfaces/interface/statistics/in-octets` to `decimal64` and divide all values with 1000

18.3 18.3 Extra XML

If the Yang validation succeeds and the startup configuration has been committed to the running database, a user may add "extra" XML.

There are two ways to add extra XML to running database after start. Note that this XML is "merged" into running, not "committed".

The extra-xml feature is not available if startup mode is *none*. It will also not occur in failsafe mode.

18.3.1 18.3.1 Via file

The first way is via a file. Assume you want to add this xml:

```
<config>
  <x xmlns="urn:example:clixon">extra</x>
</config>
```

You add this via the -c option:

```
clixon_backend ... -c extra.xml
```

18.3.2 18.3.2 Reset callback

The second way is by programming the plugin_reset() in the backend plugin. The following code illustrates how to do this (see also example_reset() in example_backend.c):

```
int
example_reset(clixon_handle h,
              const char   *db)
{
   cxobj *xt = NULL;
   yang_stmt *yspec;

   yspec = clicon_dbspec_yang(h);
   /* Parse extra XML */
   if (clixon_xml_parse_string("<x xmlns=\"urn:example:clixon\">extra</x>"
                               YB_MODULE, yspec, &xt, NULL) < 0)
      err;
   xml_name_set(xt, "config");
   /* Write to db */
   if (xmldb_put(h, (char*)db, OP_MERGE, xt, NULL, NULL) < 0)
      err;
```

(continues on next page)

(continued from previous page)

```
}
static clixon_plugin_api api = {
    ...
    .ca_reset=example_reset,
    ...
}
```

The `example_reset` function is registered in the plugin init code and is then called with an empty temp database (db). The code writes the extra XML into db (`xmldb_put`).

After exit of the callback, the system merges the temporary db into the running datastore in the same way as via file, ie not via a commit.

18.4 18.4 Failsafe mode

If the startup fails, the backend looks for a *failsafe* configuration in `<CLICON_XMLDB_DIR>/failsafe_db`. If such a config is not found, the backend terminates. In this mode, running and startup mode are unchanged.

If the failsafe is found, the running-db is copied to tmp-db and the failsafe config is loaded and committed into the running db.

If the startup mode was *startup*, the *startup* database will contain syntax errors or invalidated XML.

If the startup mode was *running*, the the *tmp* database will contain syntax errors or invalidated XML.

18.5 18.5 Repair

If the system is in failsafe mode (or fails to start), a user can repair a broken configuration and then restart the backend. This can be done out-of-band by editing the startup db and then restarting clixon.

In some circumstances, it is also possible to repair the startup configuration on-line without restarting the backend. This section shows how to repair a startup datastore on-line.

However, on-line repair *cannot* be made in the following circumstances:

- The broken configuration contains syntactic errors - the system cannot parse the XML.
- The startup mode is *running*. In this case, the broken config is in the *tmp* datastore that is not a recognized Netconf datastore, and has to be accessed out-of-band.
- Netconf must be used. Restconf cannot separately access the different datastores.

First, copy the (broken) startup config to candidate. This is necessary since you cannot make *edit-config* calls to the startup db:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <copy-config>
    <source><startup/></source>
    <target><candidate/></target>
  </copy-config>
</rpc>
```

You can now edit the XML in candidate. However, there are some restrictions on the edit commands. For example, you cannot access invalid XML (eg that does not have a corresponding module) via the edit-config operation. For example, assume *x* is obsolete syntax, then this is *not* accepted:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target><candidate/></target>
    <config>
      <x xmlns="example" operation='delete'/>
    </config>
  </edit-config>
</rpc>
```

Instead, assuming *y* is a valid syntax, the following operation is allowed since *x* is not explicitly accessed:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target><candidate/></target>
    <config operation='replace'>
      <y xmlns="example"/>
    </config>
  </edit-config>
</rpc>
```

Finally, the candidate is validate and committed:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>
```

The example shown in this Section is also available as a regression repair test script.

18.6 18.6 Automatic upgrades

There is an EXPERIMENTAL xml changelog feature based on "draft-wang-netmod-module-revision-management-01" (Zitao Wang et al) where changes to the Yang model are documented and loaded into Clixon. The implementation is not complete.

When upgrading, the system parses the changelog and tries to upgrade the datastore automatically. This feature is experimental and has several limitations.

You enable the automatic upgrading by registering the changelog upgrade method in `clixon_plugin_init()` using wildcards:

```
upgrade_callback_register(h, xml_changelog_upgrade, NULL, 0, 0, NULL);
```

The transformation is defined by a list of changelogs. Each changelog defined how a module (defined by a namespace) is transformed from an old revision to a new. Example from auto upgrade test script:

```
<changelogs xmlns="http://clicon.org/xml-changelog">
  <changelog>
    <namespace>urn:example:b</namespace>
    <revfrom>2017-12-01</revfrom>
    <revision>2017-12-20</revision>
    ...
  <changelog>
</changelogs>
```

Each changelog consists of set of (ordered) steps:

```
<step>
  <name>1</name>
  <op>insert</op>
  <where>/a:system</where>
  <new><y>created</y></new>
</step>
<step>
  <name>2</name>
  <op>delete</op>
  <where>/a:system/a:x</where>
</step>
```

Each step has an (atomic) operation:

- rename - Rename an XML tag
- replace - Replace the content of an XML node
- insert - Insert a new XML node
- delete - Delete and existing node
- move - Move a node to a new place

A *step* has the following arguments:

- where - An XPath node-vector pointing at a set of target nodes. In most operations, the vector denotes the target node themselves, but for some operations (such as insert) the vector points to parent nodes.
- when - A boolean XPath determining if the step should be evaluated for that (target) node.

Extended arguments:

- tag - XPath string argument (rename)
- new - XML expression for a new or transformed node (replace, insert)
- dst - XPath node expression (move)

Step summary:

- rename(where:targets, when:bool, tag:string)
- replace(where:targets, when:bool, new:xml)
- insert(where:parents, when:bool, new:xml)
- delete(where:parents, when:bool)
- move(where:parents, when:bool, dst:node)

CHAPTER

NINETEEN

19 ERRORS AND DEBUG

19.1 19.1 Error reporting

19.1.1 19.1.1 Initialization

Clixon core applications typically have a command-line option controlling the logs as follows:

-l <option> Log on (s)yslog, std(e)rr, std(o)ut or (f)ile. Syslog is default. If foreground, then syslog and stderr is default. Filename is given after -f as follows: `-lf<file>`.

An example of a clixon error as it may appear in a syslog:

```
Mar 24 10:30:48 Alarik clixon_restconf[3993]: clixon_restconf openssl: 3993 Started
```

In C-code, clixon error and logging is initialized by `clixon_log_init`:

```
clixon_log_init(h, prefix, upto, flags);
```

where:

- *prefix*: appears first in the error string
- *upto*: log priority as defined by syslog(3), eg: LOG_DEBUG, LOG_INFO,..
- *flags*: a bitmask of where logs appear, values are: `CLIXON_LOG_STDERR`, `_STDOUT`, `_SYSLOG`, `_FILE`.

19.1.2 19.1.2 Error call

An error is typically called by `clixon_err()` and a return value of `-1` as follows:

```
clixon_err(category, errno, format, ...)
return -1;
```

where:

- *category* is an error "category" including for example "yang", "xml" See *enum clixon_err* for more examples.
- *errno* if given, usually errors as given by `errno.h`
- *format* A variable arg string describing the error.

19.1.3 19.1.3 CLI Errors

There are several types of error messages in the CLI. The first class is "syntax" errors with things like:

```
cli> command
CLI syntax error: "foo": Unknown command
cli>
```

These are errors immediately detected by the CLIgen parser and are internally generated in CLIgen. Errors include command, syntax and type checking. They are shown on stderr, the CLI continues, without logging.

A second type of errors are "semantic" errors detected when processing CLI callbacks. These errors are more heavy-weight than syntax errors and are declared in code using standard clixon *Error call*. They are logged and can be directed to syslog, and are by default printed on stderr. The CLI continues after the error message is printed. Typical places are user callbacks, backend rpc errors, validation, etc, either system-defined or user-defined callbacks. They are on the form:

```
cli> command
Nov 15 15:42:56: acl_get_list: 334: Yang error: no ACLs defined
CLI command error
cli>
```

A third class of CLI errors are similar to the previous class but quits the CLI:

```
cli> command
Nov 15 15:42:56: acl_get_list: 334: Yang error: no ACLs defined
sh#
```

These errors are typically due to system functions failing in a fatal way.

19.1.4 19.1.4 Error categories

An application can specialize error handling for a specific category by using *clixon_err_cat_reg()* and a log callback. Example:

```
/* Clixon error category log callback
 * @param[in]    handle  Application-specific handle
 * @param[out]   cb      Read log/error string into this buffer
 */
static int
my_log_cb(void  *handle,
          cbuf  *cb)
{
    cprintf(cb, "Myerror");
    return 0;
}

main(){
  ...
  /* Register error callback for category */
  clixon_err_cat_reg(OE_SSL, h, openssl_cat_log_cb);
```

In this example, "Myerror" will appear in the log string.

19.2 19.2 Debugging

19.2.1 19.2.1 Debug flags

Each clixon application has a `-D <level>` command-line option to enable debug flags when starting a program. Levels can be combined and use either symbolic or numerical values. Example:

```
clixon_cli -D default -D detail
```

Levels are separated into *subject-area* and *detail*.

The subject-area levels are the following:

- `default` Default logs
- `msg` In/out messages and datastore reads
- `init` Initialization
- `xml` XML logs
- `xpath` XPath processing logs
- `yang` YANG processing logs
- `backend` Backend-specific processing
- `cli` CLI-frontend
- `netconf` Netconf-frontend
- `restconf` Restconf-frontend
- `snmp` SNMP-frontend
- `nacm` Netconf access control model
- `proc` Process handling
- `datastore` Datastore handling
- `event` Event handling
- `rpc` RPC handling
- `rpc` Notification streams
- `app` Application-specific handling, ie any application using clixon can use this
- `app2` Application-specific 2
- `app3` Application-specific 3
- `all` All subject-area flags

The detail area levels are the following:

- `detail` Detail logging
- `detail2` Extra details
- `detail3` Probably more detail than you want

You can combine flags, so that, for example `-D 5` means default + detailed, but no packet debugs. Similarly, some messages require multiple flags, like XML + DETAIL would be `-D 20`.

You can direct the debug logs using the `-l <option>` as follows:

- `s` : syslog
- `e` : stderr
- `o` : stdout
- `n` : none
- `f` : file, followed by a filename, eg `-f/tmp/foo`

Example:

```
clixon_backend -D 5 -f/tmp/log.txt
```

19.2.2 19.2.2 Change debug

You can also change debug level in run-time in different ways. For example, using netconf to change debug level in backend:

```
echo "<rpc username=\"root\" xmlns=\"urn:ietf:params:xml:ns:netconf:base:1.0\"><debug␣
↪xmlns=\"http://clicon.org/lib\"><level>1</level></debug></rpc>]]>]]>" | clixon_netconf␣
↪-q0
```

In this example, netconf is run using EOM encoding and does not require hello:s.

Using curl to change debug in backend via the restconf daemon:

```
curl -Ssik -X POST -H "Content-Type: application/yang-data+json" http://localhost/
↪restconf/operations/clixon-lib:debug -d '{"clixon-lib:input":{"level":1}}'
```

19.2.3 19.2.3 Debugger

Enable debugging when configuring (compile-time):

```
./configure --enable-debug
```

which includes symbol table info so that you can make breakpoints on functions(output is omitted):

```
> sudo gdb clixon_backend
(gdb) run -FD 1 -l e
Starting program: /usr/local/sbin/clixon_backend -FD 1 -l e
(gdb) b main
Breakpoint 1 at 0x55555555bcea: file backend_main.c, line 492.
(gdb) where
#0  main (argc=5, argv=0x7fffffffe4e8) at backend_main.c:492
```

In the example, the backend runs in the foreground(*-F*), runs with debug level *1* and directs the debug messages to stderr.

19.2.4 19.2.4 Valgrind and callgrind

Examples of using valgrind for memeory checks:

```
valgrind --leak-check=full --show-leak-kinds=all clixon_netconf -qf /tmp/myconf.xml -y /
↪tmp/my.yang
```

Example of using callgrind for profiling:

```
LD_BIND_NOW=y valgrind --tool=callgrind clixon_netconf -qf /tmp/myconf.xml -y /tmp/my.
↪yang
sudo kcachegrind
```

Or massif for memory usage:

```
valgrind --tool=massif clixon_netconf -qf /tmp/myconf.xml -y /tmp/my.yang
massif-visualizer
```

19.3 19.3 Customization

Errors, logs and denugs can be customized by plugins via the *ca_errmsg* API.

Customized errors applies to all clixon applications. For example, logs for the backend and return output in the CLI.

The API provides a single function callback which can be used in a various ways. The example shows one simple way as described here.

First define an error message callback as part of the plugin initialization:

```
static clixon_plugin_api api = {
  ...
 .ca_errmsg=example_cli_errmsg,
};
```

The errmsg callback has many parameters. Some are not always applicable:

- h : Clixon handle
- fn : name of source file (only err)
- line: line of source file (only err)
- type: log, err or debug (actual types called `LOG_TYPE_LOG` etc)
- category: Error category (see Section *Error categories*) (only err)
- suberr: Error number, eg `errno` (only err)
- xerr: XML structure, either NETCONF (for err) or just generic XML (debug, log)
- format: Format string similar to *printf*
- ap: Variable argument list assciated with format. Similar to *vprintf*
- cbmsg: Customized error message as output of the function. If NULL, use regular message.

A simple way to replace all error messages would be:

```
int
example_cli_errmsg(clixon_handle        h,
                   const char          *fn,
                   const int            line,
                   enum clixon_log_type type,
                   int                 *category,
                   int                 *suberr,
                   cxobj               *xerr,
                   const char          *format,
                   va_list              ap,
                   cbuf                **cbmsg)
{
    if (type != LOG_TYPE_ERR)
       return 0;
    if ((*cberr = cbuf_new()) == NULL){
       fprintf(stderr, "cbuf_new: %s\n", strerror(errno));
       return -1;
    }
    cprintf(*cberr, "My error message");
    *category = 0;
    suerr = 0;
    retval = 0;
 done:
    return retval;
}
```

All error message are now:

```
My error message
```

Which may not be useful.

More logic needs to be added, for example a more advanced classification and translation/changing of error messages. Any field can be used to classify. The *format* string and the *ap* objects may be translated/converted which is out-of-scope of this document.

19.3.1 19.3.1 Indirection

The customized callback may also be changed dynamically. The example shows an extra indirection layer, where a new function is registered before a call, and deregistered after.

Please see the main example, where *example_cli_errmsg* just dispatches the call to a dynamic *myerrmsg*.

CHAPTER

TWENTY

20 MISC

These are sections that do not fit into the rest of the document.

20.1 20.1 High availability

This is a brief note on a potential future feature.

Clixon is mainly a stand-alone app tightly coupled to the application/device with "shared fate", that is, if clixon fails, so does the application.

That said, the primary state is the *backend* holding the *configuration database* that can be shared in several ways. This is not implemented in Clixon, but potential implementation strategies include:

- *Active/standby*: With a standard failure/liveness detection of a master backend, a standby could be started when the master fails using "-s running" (just picking up the state from the failed master). The default cache write-through can be used (`CLICON_DATASTORE_CACHE = cache`). Would suffer from outage during standby boot.
- *Active/active*: The config-db cache is turned off (`CLICON_DATASTORE_CACHE = nocache`) and two backend process started with a load-balancing in front. Turning the cache off would suffer from performance degradation (and its not currently tested in regression tests). Would also need a failure/liveness detection.

In both cases the *config-db* would be a single-point-of-failure but could be mitigated by a replicated file system, for example.

Regarding clients:

- the *CLI* and *NETCONF* clients are stateless and spun up on demand.
- the *RESTCONF* daemon is stateless and can run as multiple instances (with a load-balancer)

20.2 20.2 Process handling

Clixon has a simple internal process handling currently used for *internal restconf* but can also be used for user applications. The process data structure has a unique name with pids created for "active" processes. There are three states:

STOPPED
: pid=0, No process running

RUNNING
: pid is set, Process started and assumed running

EXITING
pid set, Process is killed by parent but not waited for (eg not

There are three operations that a client can perform on processes:

start
Start a process

restart
Restart a process

stop
Stop a process

20.2.1 20.2.1 State machine

The state machine for a process is as follows:

```
--> STOPPED --(re)start-->    RUNNING(pid)
       ^   <--1.wait(kill)---   |  ^
       |                   stop/|  |
       |                 restart|  | restart
       |                        v  |
       wait(stop) ------- EXITING(dying pid)
```

The Process struct is created by calling clixon_process_register() with static info such as name, description, namespace, start arguments, etc. Starts in STOPPED state:

```
--> STOPPED
```

On operation "start" or "restart" it gets a pid and goes into RUNNING state:

```
STOPPED -- (re)start --> RUNNING(pid)
```

When running, several things may happen:

1. It is killed externally: the process gets a SIGCHLD triggers a wait and it goes to STOPPED:

   ```
   RUNNING  --sigchld/wait-->  STOPPED
   ```

2. It is stopped due to a rpc or configuration remove: The parent kills the process and enters EXITING waiting for a SIGCHLD that triggers a wait, therafter it goes to STOPPED:

   ```
   RUNNING --stop--> EXITING  --sigchld/wait--> STOPPED
   ```

3. It is restarted due to rpc or config change (eg a server is added, a key modified, etc). The parent kills the process and enters EXITING waiting for a SIGCHLD that triggers a wait, therafter a new process is started and it goes to RUNNING with a new pid:

   ```
   RUNNING --restart--> EXITING  --sigchld/wait + restart --> RUNNING(pid)
   ```

20.3 20.3 Event notifications

Clixon implements RFC 5277 NETCONF Event Notifications

The main example illustrates an EXAMPLE stream notification that triggers every 5s. First, declare a notification in YANG:

```
notification event {
     description "Example notification event.";
     leaf event-class {
       type string;
       description "Event class identifier.";
     }
     ...
```

To start a notification stream via netconf:

```
<rpc><create-subscription xmlns="urn:ietf:params:xml:ns:netmod:notification"><stream>
↪EXAMPLE</stream></create-subscription></rpc>]]>]]>
<rpc-reply><ok/></rpc-reply>]]>]]>
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0"><eventTime>2019-01-
↪02T10:20:05.929272</eventTime><event><event-class>fault</event-class><reportingEntity>
↪<card>Ethernet0</card></reportingEntity><severity>major</severity></event></
↪notification>]]>]]>
```

This can also be triggered via the CLI:

```
clixon_cli -f /usr/local/etc/example.xml
cli> notify
cli> event-class fault;
reportingEntity {
  card Ethernet0;
}
severity major;

cli> no notify
cli>
```

Restconf notifications (FCGI only) is also supported,