clixon

Release 4.0

Sep 17, 2019

Table of contents

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

CHAPTER 1

Overview

Clixon is a management framework that can be 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.

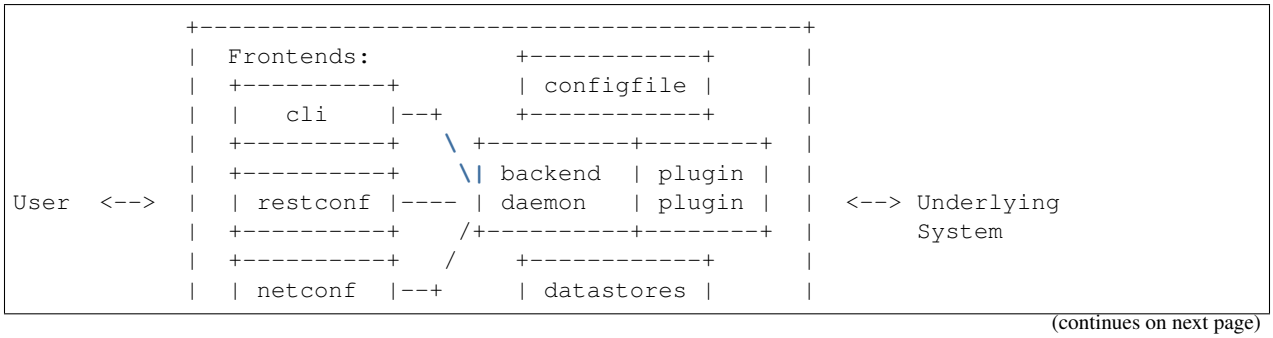
Most of the projects using Clixon are for networking devices. But Clixon can be used for other YANG-based systems as well due to a modular and pluggable architecture.

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.

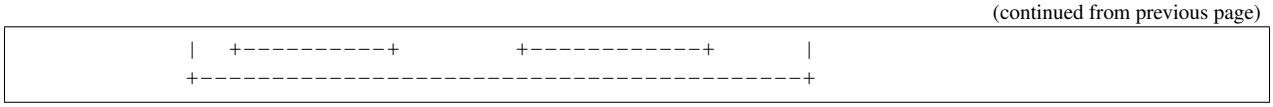
Clixon is open-source and dual licensed. Either Apache License, Version 2.0 or GNU General Public License Version 2.

1.1 System Architecture



(continues on next page)

(continued from previous page)



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

The clients provide frontend interfaces to users, including an interactive CLI, RESTCONF over HTTP, and XML NETCONF over SSH. Internally, the clients and backend communicate via NETCONF over a UNIX socket.

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

When you adapt Clixon to your system, you typically start with a set of YANG specifications that you want implemented. You then write backend plugins that interact with the underlying 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.

1.2 Platforms

Clixon supports GNU/Linux, FreeBSD and Docker. MacOS may work. Linux platforms include 32 and 64 bits Ubuntu, Alpine, Raspian, etc.

1.3 Standards

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

1.4 How to get Clixon

Get the Clixon source code from Github

1.5 Support

For support issues use the Clixon slack channel

1.6 Bug reports

Report bugs via Github issues

CHAPTER 2

Installation

Clixon runs on Linux, FreeBSD port and Mac/Apple. CPU architecures include x86_64, i686, ARM32.

You can also run Clixon in a docker container.

2.1 Ubuntu Linux

2.1.1 Installing dependencies

Install packages:

```
sudo apt-get update
sudo apt-get install flex bison fcgi-dev curl-dev
```

Install and build CLIgen:

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

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

```
sudo useradd clicon
sudo usermod -a -G clicon <user>
sudo usermod -a -G clicon www-data
```

Or using adduser:

```
sudo adduser -D -H clicon
sudo adduser <user> clicon
```

2.1.2 Build from source

```
configure                  # Configure clixon to platform
make                       # Compile
sudo make install          # Install libs, binaries, and config-files
sudo make install-include # Install include files (for compiling)
```

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 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 Build from source

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 Systemd

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

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 Systemd restconf

The Restconf service is installed at /etc/systemd/system/example_restconf.service, for example.

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

2.4 Advanced install

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

--disable-stdyangs Include standard yang files in clixon install, default: yes

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

--with-cligen=dir Use CLIGEN here

--without-restconf disable support for restconf

--with-wwwuser=<user> Set www user different from www-data

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

--with-libxml2 use gnome/libxml2 regex engine

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

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

CHAPTER 3

Quick start

This section describes how to run the Hello world example available in the soure code at: 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 specification and a CLI specification.

A more advanced application will have backend (and frontend) plugins to add application-specific semantics. This is not necessary in the hello world application.

3.1 Content

The hello example directory contains the following files:

hello.xml The configuration file. See clixon-config.yang for the uang spec of hello.xml.

clixon-hello@2019-04-17.yang The yang spec of the example.

hello_cli.cli CLIgen specification.

Makefile.in Example makefile where plugins are built and installed

README.md This file

3.2 Compile and run

Before you start, go through the install instructions.

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

Start backend in the background:

```
sudo clixon_backend
```

Start cli:

```
clixon_cli
```

3.3 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
cli> commit
cli> quit
olof@vandal>
```

3.4 Netconf

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

```
olof@vandal> clixon_netconf -q
<rpc><edit-config><target><candidate/></target><config><hello xmlns="urn:example:hello
↪"><world/></hello></config></edit-config></rpc>]]>]]>
<rpc-reply><ok/></rpc-reply>]]>]]>
<rpc><commit/></rpc>]]>]]>
<rpc-reply><ok/></rpc-reply>]]>]]>
<rpc><get-config><source><running/></source></get-config></rpc>]]>]]>
<rpc-reply><data><hello xmlns="urn:example:hello"><world/></hello></data></rpc-reply>
↪]]>]]>
olof@vandal>
```

3.5 Restconf

Clixon also provides a Restconf interface. A reverse proxy needs to be configured.

For example, using nginx on an Ubuntu release: install, and edit config file */etc/nginx/sites-available/default*

```
server {
  ...
  location /restconf {
    fastcgi_pass unix:/www-data/fastcgi_restconf.sock;
    include fastcgi_params;
  }
}
```

Start nginx daemon

```
sudo /etc/init.d/nginx start
```

or using systemd:

```
sudo systemctl start nginx.service
```

Start the restconf daemon

```
sudo su -c "/www-data/clixon_restconf" -s /bin/sh www-data &
```

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

3.6 Run a container

You can run the hello example as a pre-built docker container, on a *x86_64* Linux. First, the container is started with the backend running:

```
docker run --rm --name hello -d clixon/clixon clixon_backend -Fs init
```

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
<rpc><get-config><source><candidate/></source></get-config></rpc>]]>]]>
<rpc-reply><data/></rpc-reply>]]>]]>
```

3.7 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 4

Standards

4.1 YANG

YANG and XML is the heart of Clixon. Yang modules are used as a specification for handling XML configuration data. The YANG spec is used to generate an interactive CLI, netconf and restconf clients. It also specifies the format of the XML datastore.

The YANG standards that Clixon follows include:

- YANG 1.0 RFC 6020
- YANG 1.1 RFC 7950
- RFC 7895: YANG module library

However, the following YANG syntax modules are not implemented (reference to RFC7950 in parenthesis):

- deviation (7.20.3)
- action (7.15)
- augment in a uses sub-clause (7.17) (module-level augment is implemented)
- require-instance
- instance-identifier type (9.13)
- status (7.21.2)
- YIN (13)
- Yang extended Xpath functions: re-match(), deref)(), derived-from(), derived-from-or-self(), enum-value(), bit-is-set() (10.2-10.6)
- Default values on leaf-lists (7.7.2)
- Lists without keys (non-config lists may lack keys)

4.1.1 Regular expressions

Clixon supports two regular expressions engines:

Posix Posix is the default method, _translates_ XSD regexp:s 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>
```

4.2 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

The following XPATH axes are supported:

- child,
- descendant,
- descendant_or_self,
- self
- parent

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

4.3 NETCONF

Clixon implements the following NETCONF RFC:s:

- RFC 6241: NETCONF Configuration Protocol
- RFC 6242: Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 6243: With-defaults Capability for NETCONF. Clixon implements "explicit" default handling, but does not implement the RFC.
- RFC 5277: NETCONF Event Notifications
- RFC 8341: Network Configuration Access Control Model

The following RFC6241 capabilities/features are hardcoded in Clixon:

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

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

- :startup (RFC6241 8.7)

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

4.4 RESTCONF

Clixon Restconf is a daemon based on FastCGI C-API. Instructions are available to run with NGINX. The implementation is based on 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” and “stop-time”.
- Monitoring (Sec 9)

The following features are not implemented:

- ETag/Last-Modified
- Query parameters: “fields”, “filter”, “with-defaults”

CHAPTER 5

Configuration

The clixon configuration file is an XML file modelled by YANG. By default, it is installed in */usr/local/etc/clixon.xml*. The YANG specification for the configuration is clixon-config.yang. All Clixon processes (backend, cli, netconf, restconf) use the same config file, although some configuration options are not valid for all processes.

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

5.1 Example

The following is the configuration file of the *hello world* example:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_FEATURE>*:*</CLICON_FEATURE>
  <CLICON_CONFIGFILE>/usr/local/etc/example.xml</CLICON_CONFIGFILE>
  <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>
  <CLICON_MODULE_LIBRARY_RFC7895>false</CLICON_MODULE_LIBRARY_RFC7895>
</clixon-config>
```

5.2 Specification

Some options (of approximately 50) described below of clixon-config.yang are the following (descriptions are skipped):

```
container clixon-config {
    leaf CLICON_CONFIGFILE{
        type string;
    }
    leaf CLICON_YANG_MAIN_DIR {
        type string;
    }
    leaf-list CLICON_FEATURE {
       type string;
    }
    leaf CLICON_YANG_REGEXP {
        type regexp_mode;
        default posix;
    }
```

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

5.3 Finding the configuration

There are several ways to change where Clixon finds it config file (FILE), in priority order:

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:

   ```
   ./configure -f 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*

Note that the configure file itself may have the option *CLICON_CONFIGFILE* but due to bootstrapping reasons, its value is meaningless but can be useful for documentation purposes:

```
<CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
```

5.4 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
```

5.5 Features

CLICON_FEATURE is a list of values, describing how Clixon supports feature. Clixon have three hardcoded features:

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

Supplying the *-o* option will add a value to the list (not replace existing). For example, if *-o CLICON_FEATURE=ietf-netconf:startup* is given at startup, the following options would be present in the configuration:

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

(Which does not really make sense since **:** enables all features anayway.)

5.6 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 the module *clixon-hello* should be 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.

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.

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:

- *CLICON_YANG_MAIN_FILE*
- *CLICON_YANG_MODULE_MAIN*
- *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.

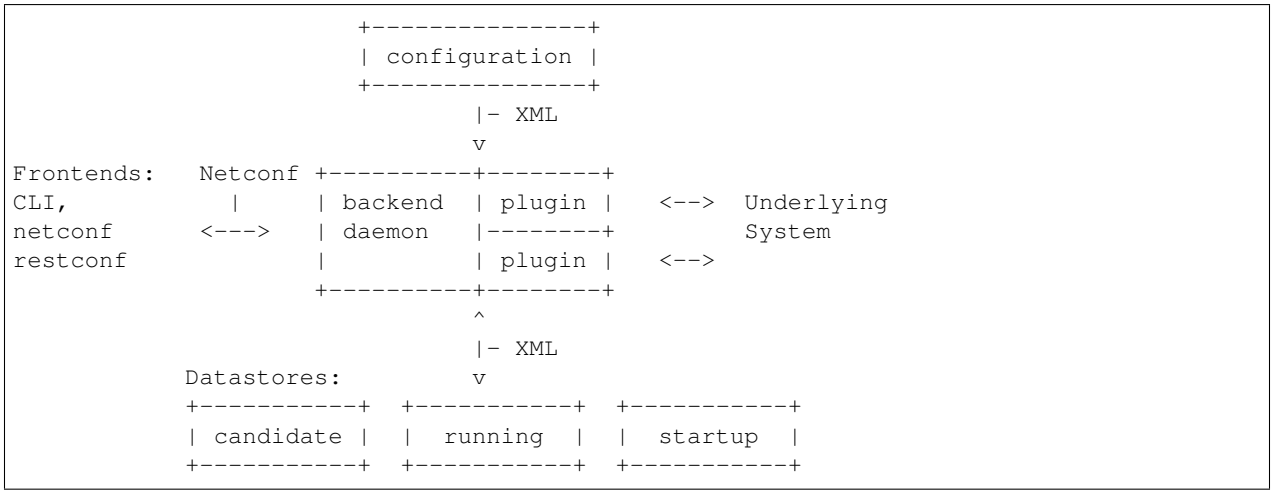
5.7 Default values

CLICON_YANG_REGEXP which is not present in the *hello world* is an example of a configuration option with a default value of *posix*:

```
<CLICON_YANG_REGEXP>posix</CLICON_YANG_REGEXP>
```

CHAPTER 6

Backend



The backend daemon is the central Clixon component. 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 A NETCONF socket to frontend clients. This is by default a UNIX domain socket but can also be an IPv4 or IPv6 TCP socket.

Datastores XML 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 the plugins.

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

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 uses a datastore cache.

6.1 Command-line options

The backend have the following command-line options:

-h	Help
-D <level>	Debug level
-f <file>	CLICON config file
-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: -lf<file>.
-d <dir>	Specify backend plugin directory (default: none)
-p <dir>	Yang directory path (see CLICON_YANG_DIR)
-b <dir>	Specify XMLDB database 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)(default: /usl/var/hello.sock)
-P <file>	PID filename (default: /usr/local/var/hello.pidfile)
-1	Run once and then quit (dont wait for events)
-s <mode>	Specify backend startup mode: none|startup|running|init)
-c <file>	Load extra xml configuration, but don't commit.
-g <group>	Client membership required to this group (default: clicon)
-U <user>	Run backend daemon as this user AND drop privileges permanently
-y <file>	Load yang spec file (override yang main module)
-o <option=value>	Give configuration option overriding config file (see clixon-config.yang)

6.1.1 Logging and debugging

At debug, the backend can be run in the foreground and with debug flags:

```
clixon_backend -FD 1
```

Logging is done 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
```

6.1.2 Startup modes

There are four different backend startup modes selected by the *-s* option. The difference is in how the running state is handled, ie what state the machine is 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.

You use the *-s* to select startup mode:

```
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>
```

6.2 Socket

```
Frontends:    socket   +----------+
CLI,            |      | backend  |
netconf       <--->    | daemon   |
restconf               |          |
                       +----------+
```

The Clixon backend creates a socket that the frontends can connect to. Communication is made over this 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 socket.

CLICON_SOCK If family above is AF_UNIX: Unix socket for communicating with clixon_backend. If family is AF_INET: 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*.

6.3 Backend files

A couple of config options control files related to the backend, as follows:

CLICON_BACKEND_DIR Location of backend .so plugins. Load all *.so* plugins in this dir as backend plugins

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

CLICON_BACKEND_PIDFILE Process-id file of backend daemon

6.4 Plugins

Backend plugins are the "glue" that binds the Clixon system to the underlying system. The backend invokes *callbacks* in the plugins when events occur.

Plugins are written in C as dynamically loaded modules (*.so* files). At startup, the backend daemon looks in the directory pointed to by the config option *CLICON_BACKEND_DIR*, and loads all files with *.so* suffixes from that dir in alphabetical order.

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

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

You can filter which plugins to load by specifying a regular expression. For example, the following will only load backend plugins starting with "example":

```
<CLICON_BACKEND_REGEXP>^example*.so$</CLICON_BACKEND_REGEXP>
```

A plugin must have a init function called *clixon_plugin_init*. If this function does not exist, the backend will fail.

The backend calls *clixon_plugin_init* and expects it to return an API struct defining all callbacks. The init function may return *NULL* in which case the backend logs this and continues.

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

6.4.1 Callbacks

The following callbacks are defined for backend plugins:

init Clixon plugin init function, called immediately after plugin is loaded into the backend. 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", (almost) all initialization is complete and daemon is in the background. If daemon privileges are dropped (see *dropping privileges*) this callback is called *before* privileges are dropped.

exit Called just before plugin is unloaded

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.

reset Reset system status

trans_begin, trans_validate, trans_complete, trans_commit, trans_revert, trans_end, trans_abort Transaction callbacks which are invoked for two reasons: validation requests or commits. These callbacks are further described in *transactions* section.

6.5 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 incrementaly upgrade a system state based on configuration changes. Writing commit callbacks is the core functionality of a 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. If you have two plugins, for example, a transaction sequence looks like the following:

```
Backend   Plugin1    Plugin2
|          |          |
+--------->+--------->+ begin
|          |          |
+--------->+--------->+ validate
|          |          |
+--------->+--------->+ commit
|          |          |
+--------->+--------->+ 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
```

6.6 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>
```

6.6.1 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 *Installation*.

6.6.2 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 the be able to raise privileges when needed:

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

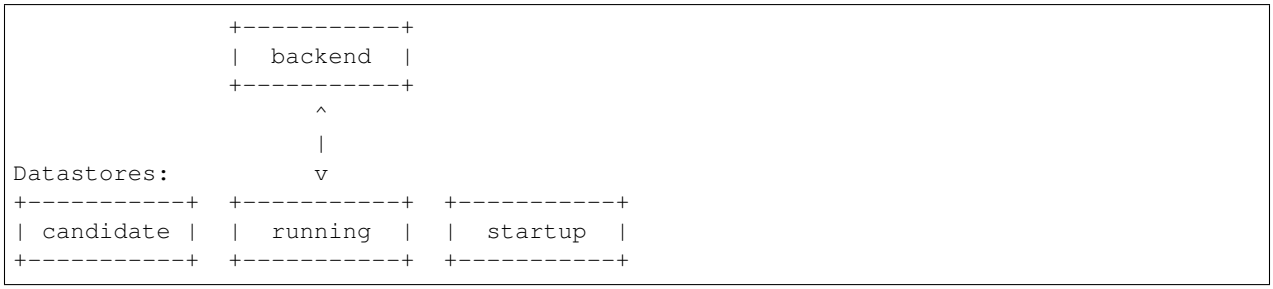
In this mode, a plugin callback (eg commit), can temporarily raise the privileges when accessing system resources, and the lower them when done.

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 7

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.

Note that there may appear other datastores, Clixon is not limited to the three datastores above. For example, a *tmp* datastore appears in several cases as an intermediate datastore.

7.1 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*) 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
-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*.

7.2 Datastore format

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 only fully supported alternative. *json* and *tree* are experimental (the latter is a random-access binary format).

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.

7.3 Module library support

Clixon can store Yang module-state information according to RFC 7895: YANG module library in the datastores. With module state, you know which Yang version the XML belongs to, and is useful when upgrading, for example.

Including yang module-state in the datastores is enabled by the following entry in the Clixon configuration:

```
<CLICON_XMLDB_MODSTATE>true</CLICON_XMLDB_MODSTATE>
```

A backend performs detection of mismatching XML/Yang if *CLICON_XMLDB_MODSTATE* was not enabled when saving the file so that it contains module-state information; and the backend configuration has *CLICON_XMLDB_MODSTATE* enabled.

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

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

7.4 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.

CHAPTER 8

CLI

The CLI using CLIgen is a central part of Clixon.

8.1 Using the CLI

Once the backend is started, the easiest way to use Clixon is via the CLI.

The following example shows how to add an interface in candidate, validate and commit it to running, then look at it (as xml) and finally delete it:

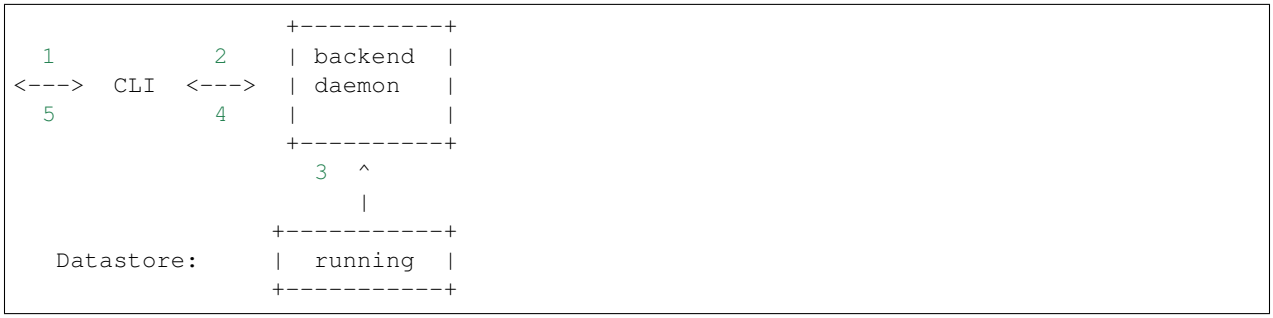
```
  clixon_cli -f /usr/local/etc/example.xml
  cli> set interfaces interface eth9 ?
    description               enabled                   ipv4
    ipv6                      link-up-down-trap-enable  type
  cli> set interfaces interface eth9 type ex:eth
  cli> validate
  cli> commit
  cli> show configuration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

CHAPTER 9

Usecases

This section contains usescases which illustrate the flow of data from a user via Clixon frontends, backend to the underlying system and back.

9.1 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 *Quick start*). 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:

```
<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.

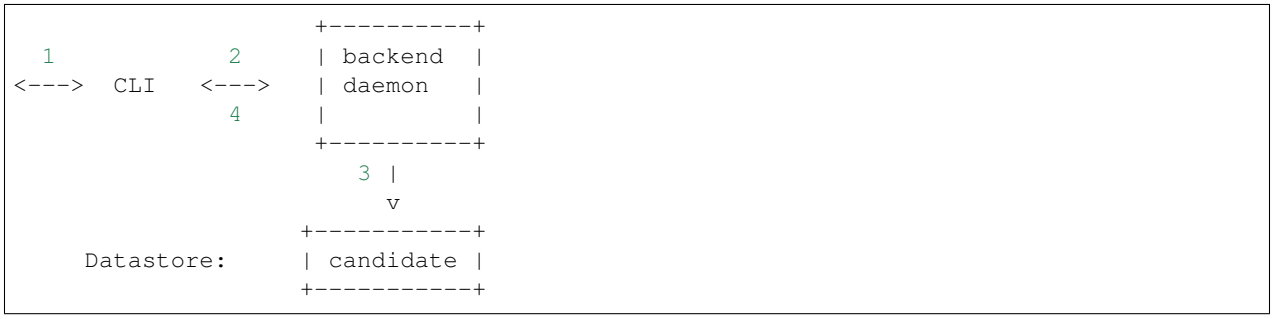
4. The backend returns the filtered output to the client:

```
<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).

9.2 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 commited to running which triggers a plugin commit transaction:

1. CLI example command:

```
cli> set hello world
cli>
```

2. Internal netconf containing a "replace" operation:

```
<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>
  </edit-config>
</rpc>
```

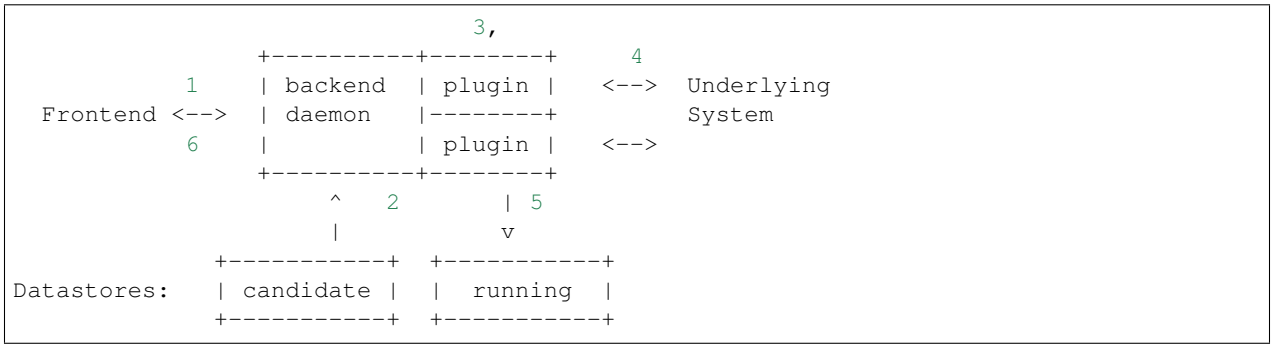
3. The backend modifies the *candidate* datastore. If there was no previous content it will look like the following after the edit:

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

4. The backend will reply with an OK:

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

9.3 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):

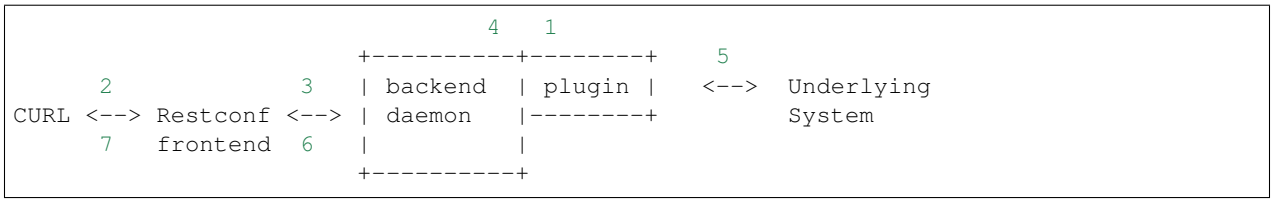
```
<rpc username="olof">
  <commit/>
</rpc>
```

2. When the backend receives the commit message, it computes the differences between candidate and running datastores, creates a transaction datastructure and initites 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>
  <ok/>
</rpc-reply
```

9.4 RESTCONF RPC



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>
  <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 10

Advanced

(Work in progress, these are sections that do not fit into the rest of the document)

10.1 CLI

10.1.1 Differences 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 following CLIgen 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.
  - *CLICON_PROMPT*: A string describing the CLI prompt using a very simple format with: *%H*, *%U* and *%T*.
  - *CLICON_PLUGIN*: the name of the object file containing callbacks in this file.
- 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.

10.1.2 History

Clixon CLI supports persistent command history. There are two CLI history related configuration options: *CLICON_CLI_HIST_FILE* with default value *~/.clixon_cli_history* and *CLICON_CLI_HIST_SIZE* with default value 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 will be logged at debug level

10.1.3 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* can be used to add the same syntax in multiple modes. For example, if you have major modes *configure* and *operation* and a set of commands that should be in both, you can add a sub-mode that will appear in both configure and operation mode.

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

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 (not the others):

```
cli> show <TAB>
  routing      files
```

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 $@ $<
```