Clixon-controller

Release 1.0

Olof Hagsand

Apr 03, 2024

TABLE OF CONTENTS

Clixon network controller is an open-source manager of network devices based on NETCONF and YANG.

CHAPTER

ONE

1 OVERVIEW

Th clixon network controller is an open-source manager of network devices based on NETCONF and YANG.

The controller is based on Clixon. The controller is a Clixon application.

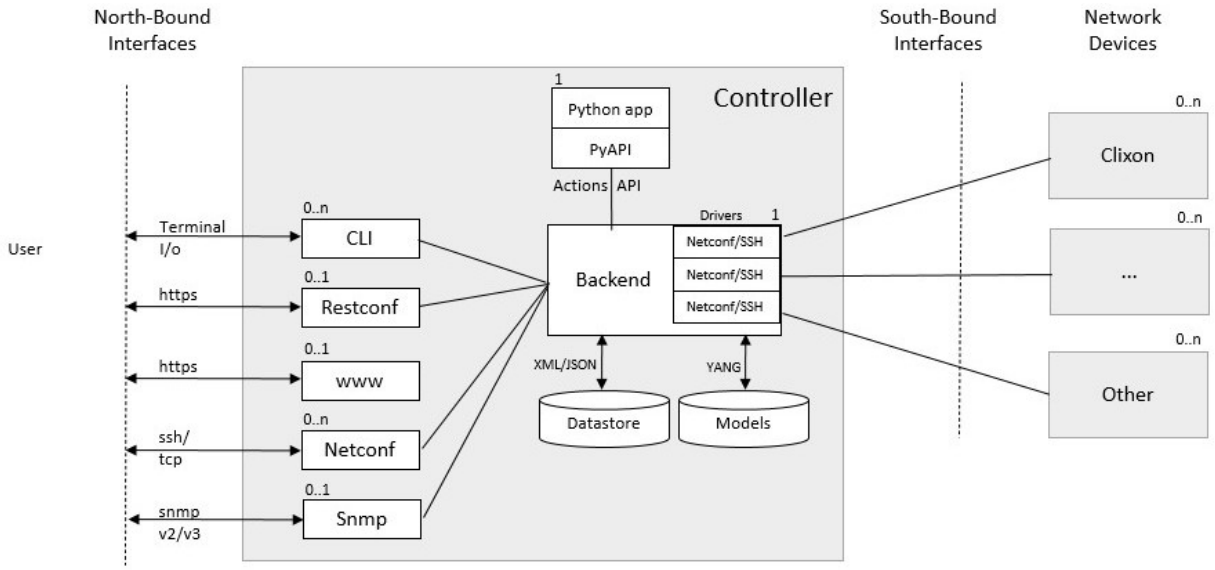
1.1 1.1 Goals

The Clixon network controller aims at providing a simple network controller for NETCONF devices of different vendors, not only Clixon.

Further goals are:

- Programmable network services, with a Python API
- Multiple devices, with different YANG schemas using RFC 8528: YANG Schema Mount .
- Transactions with validate/commit/revert across groups of devices
- Scaling up to 100 devices.

1.2 1.2 Architecture



The controller is built on the base of the CLIgen/Clixon system, where the controller semantics is implemented using plugins. The *backend* is the core of the system controlling the datastores and accessing the YANG models.

1.3 1.3 APIs

The *southbound API* uses only NETCONF over SSH to network devices. There are no current plans to support other protocols for device control.

The *northbound APIs* are YANG-derived Restconf, Autocli, Netconf, and Snmp. The controller CLI has two modes: operation and configure, with an autocli configure mode derived from YANG.

A PyAPI module accesses configuration data via the actions API. The PyAPI module reads services configuration and writes device data. The backend then pushes changes to the actual devices using a transaction mechanism.

CHAPTER

TWO

2 INSTALLATION

2.1 2.1 Packages

Some packages are required. The following are example of debian-based packages:

```
sudo apt install flex bison git make gcc libnghttp2-dev libssl-dev
```

2.2 2.2 Source

Check out the following GIT repos:

- https://github.com/clicon/cligen.git/
- https://github.com/clicon/clixon.git/
- https://github.com/clicon/clixon-controller.git/
- https://github.com/clicon/clixon-pyapi.git/

2.3 2.3 Building

The source is built as follows.

2.3.1 2.3.1 Cligen

```
cd cligen
./configure
make
sudo make install
```

2.3.2 2.3.2 Clixon

```
cd clixon
./configure
make
sudo make install
```

2.3.3 2.3.3 Python API

```
# Build and install the package
cd clixon-pyapi
sudo -u clicon pip3 install -r requirements.txt
sudo python3 setup.py install
```

2.3.4 2.3.4 Controller

```
cd clixon-controller
./configure
make
sudo make install
```

2.4 2.4 Configure options

The Controller *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

--with-cligen=dir Use CLIGEN here

--with-clixon=dir Use Clixon here

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

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

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

2.5 2.5 Python install

Install the python code by copy:

```
sudo cp clixon_server.py /usr/local/bin/
```

Add a new clicon user and install the needed Python packages, the backend will start the Python server and drop the privileges to this user:

```
sudo useradd -g clicon -m clicon
```

CHAPTER

THREE

3 QUICK START

Start example devices as containers:

```
cd test
./start-devices.sh
sudo ./copy-keys.sh
```

Start controller:

```
sudo clixon_backend -f /usr/local/etc/controller.xml
```

Start the CLI and configure devices:

```
clixon_cli -f /usr/local/etc/controller.xml -m configure
set devices device clixon-example1 description "Clixon container"
set devices device clixon-example1 conn-type NETCONF_SSH
set devices device clixon-example1 addr 172.20.20.2
set devices device clixon-example1 user root
set devices device clixon-example1 enable true
set devices device clixon-example1 yang-config VALIDATE
set devices device clixon-example1 root
commit local
```

Thereafter explicitly connect to the devices:

```
clixon_cli -f /usr/local/etc/controller.xml
connection open
```

CHAPTER

FOUR

4 CONFIGURATION

The controller extends the clixon configuration file as follows:

CLICON_CONFIG_EXTEND
: The value should be *clixon-controller-config* making the controller-specific

CONTROLLER_ACTION_COMMAND
: Should be set to the PyAPI binary with correct arguments The namespace is =”http://clicon.org/controller-config”

CLICON_BACKEND_USER
: Set to the user which the action binary (above) is used. Normally *clicon*

CLICON_SOCK_GROUP
: Set to user group, ususally *clicon*

CONTROLLER_YANG_SCHEMA_MOUNT_DIR
: Directory where device YANGs are stored locally. Both for RFC 6022 get-schema retrieval as well as local module-set YANGs.

CONTROLLER_PYAPI_MODULE_PATH
: Path to Python code for PyAPI

CONTROLLER_PYAPI_MODULE_FILTER

CONTROLLER_PYAPI_PIDFILE

4.1 4.1 Example

The following configuration file examplifies the configure options described above:

```
<clixon-config xmlns="http://clicon.org/config">
<CLICON_CONFIGFILE>/usr/local/etc/controller.xml</CLICON_CONFIGFILE>
<CLICON_FEATURE>ietf-netconf:startup</CLICON_FEATURE>
<CLICON_FEATURE>clixon-restconf:allow-auth-none</CLICON_FEATURE>
<CLICON_CONFIG_EXTEND>clixon-controller-config</CLICON_CONFIG_EXTEND>
<CONTROLLER_ACTION_COMMAND xmlns="http://clicon.org/controller-config">
      /usr/local/bin/clixon_server.py -f /usr/local/etc/controller.xml -F
</CONTROLLER_ACTION_COMMAND>
<CONTROLLER_PYAPI_MODULE_PATH xmlns="http://clicon.org/controller-config">
      /usr/local/share/clixon/controller/modules/
</CONTROLLER_PYAPI_MODULE_PATH>
<CONTROLLER_PYAPI_MODULE_FILTER xmlns="http://clicon.org/controller-config"></CONTROLLER_
↪PYAPI_MODULE_FILTER>
```

(continues on next page)

(continued from previous page)

```
<CONTROLLER_PYAPI_PIDFILE xmlns="http://clicon.org/controller-config">
      /tmp/clixon_server.pid
</CONTROLLER_PYAPI_PIDFILE>
<CLICON_BACKEND_USER>clicon</CLICON_BACKEND_USER>
<CLICON_SOCK_GROUP>clicon</CLICON_SOCK_GROUP>
```

CHAPTER

FIVE

5 CLI

This section desribes the CLI commands of the Clixon controller. A simple example is used to illustrate concepts.

5.1 5.1 General

5.1.1 5.1.1 Version

You can show the version either with the `-V` command-line option or with the CLI show version command:

```
> clixon_cli -V
Clixon version: 6.6.0
CLIgen:         6.6.0
Controller:     1.0.0
Controller GIT: 40290c0
Controller bld: 2024.02.15 13:19 by clixon on paradise
```

5.2 5.2 Modes

The CLI has two modes: operational and configure. The top-levels are as follows:

```
> clixon_cli
cli> ?
  configure             Change to configure mode
  connection            Change connection state of one or several devices
  debug                 Debugging parts of the system
  exit                  Quit
  processes             Process maintenance
  pull                  Pull config from one or multiple devices
  push                  Push config to one or multiple devices
  quit                  Quit
  save                  Save running configuration to XML file
  shell                 System command
  show                  Show a particular state of the system
  transaction           Controller transaction

cli> configure
cli[/]# set ?
```

(continues on next page)

(continued from previous page)

```
  devices               Device configuration
  processes             Processes configuration
  services              Placeholder for services
cli[/]#
```

5.3 5.3 Devices

Device configuration is separated into two domains:

1) Local information about how to access the device (meta-data)
2) Remote device configuration pulled from the device.

The user must be aware of this distinction when performing *commit* operations.

5.3.1 5.3.1 Local device configuration

The local device configuration contains information about how to access the device:

```
device clixon-example1 {
   description "Clixon example container";
   enabled true;
   conn-type NETCONF_SSH;
   user admin;
   addr 172.17.0.3;
   yang-config VALIDATE;
}
```

A user makes a local commit and thereafter explicitly connects to a locally configured device:

```
# commit local
# exit
> connection open
```

5.3.2 5.3.2 Device profile

You can configure a device *profile* that applies to severaldevices. This is useful when configuring devices of a specific vendor.

Example:

```
device-profile myprofile {
   description "Clixon example container";
   conn-type NETCONF_SSH;
   user admin;
   yang-config VALIDATE;
   module-set {
      module openconfig-interfaces {
         namespace http://openconfig.net/yang/interfaces;
      }
```

(continues on next page)

(continued from previous page)

```
   }
}
device clixon-example1 {
   device-profile myprofile;
   addr 172.17.0.3;
   enabled true;
}
device clixon-example2 {
   device-profile myprofile;
   addr 172.17.0.4;
   enabled true;
}
```

In the example, the *myprofile* device-profile defines a set of common fields, including the locally loaded openconfig YANG. See Section *YANG* for more information on loading device YANGs.

5.3.3 5.3.3 Remote device configuration

The remote device configuration is present under the *config* mount-point:

```
device clixon-example1 {
   ...
   config {
      interfaces {
         interface eth0 {
            mtu 1500;
         }
      }
   }
}
```

The remote device configuration is bound to device-specific YANG models downloaded from the device at connection time.

5.3.4 5.3.4 Device naming

The local device name is used for local selection:

```
device example1
```

Wild-cards (globbing) can be used to select multiple devices:

```
device example*
```

Further, device-groups can be configured and accessed as a single entity:

```
device-group all-examples
```

Note: Device groups can be statically configured but not used in most operations

In the forthcoming sections, selecting <*devices*> means any of the methods described here.

5.3.5 5.3.5 Device state

Examine device connection state using the show command:

```
cli> show devices
Name                    State      Time                    Logmsg
================================================================================
example1                OPEN       2023-04-14T07:02:07
example2                CLOSED     2023-04-14T07:08:06    Remote socket endpoint closed
```

There is also a detailed variant of the command with more information in XML:

```
olof@zoomie> show devices detail
<devices xmlns="http://clicon.org/controller">
  <device>
    <name>example1</name>
    <description>Example container</description>
    <enabled>true</enabled>
    ...
```

5.3.6 5.3.6 (Re)connecting

When adding and enabling one a new device (or several), the user needs to explicitly connect:

```
cli> connection <devices> connect
```

The "connection" command can also be used to close, open or reconnect devices:

```
cli> connection <devices> reconnect
```

5.4 5.4 Syncing from devices

5.4.1 5.4.1 pull

Pull fetches the configuration from remote devices and replaces any existing device config:

```
cli> pull <devices>
```

The synced configuration is saved in the controller and can be used for diffs etc.

5.4.2 5.4.2 pull merge

```
cli> pull <devices> merge
```

This command fetches the remote device configuration and merges with the local device configuration. use this command with care.

5.5 5.5 Services

Network services are used to generate device configs.

5.5.1 5.5.1 Service process

To run services, the PyAPI service process must be enabled:

```
cli# set services enabled true
cli# commit local
```

To view or change the status of the service daemon:

```
cli> service process ?
  restart
  start
  status
  stop
```

5.5.2 5.5.2 Example

An example service could be:

```
cli> set service test 1 e* 1400
```

which adds MTU *1400* to all interfaces in the device config:

```
interfaces {
  interface eth0{
    mtu 1400;
  }
  interface enp0s3{
    mtu 1400;
  }
}
```

Service scripts are written in Python using the PyAPI, and are triggered by commit commands.

You can also trigger service scripts as follows:

```
cli# apply services
cli# apply services testA foo
cli# apply services testA foo diff
```

In the first variant, all services are applied. In the second variant, only a specific service is triggered.

5.5.3 5.5.3 Created objects

The system keeps track of which device objects are created, so that they can be be removed when the service is removed. A service tags device objects with a *creator attribute* which results in a set of *created* configure objects in the controller.

The list created objects can be viewed as part of the regular configuration:

```
cli> show configuration services ssh-users test1 created
<services xmlns="http://clicon.org/controller">
   <ssh-users xmlns="urn:example:test">
      <name>test1</name>
      <created>
         <path>/devices/device[name="openconfig1"]/config/system/aaa/authentication/
↪users/user[username="test1"]</path>
         <path>/devices/device[name="openconfig2"]/config/system/aaa/authentication/
↪users/user[username="test1"]</path>
      </created>
   </ssh-users>
</services>
```

Debugging

If you enable debugging (`-D app`), an entry is logged to the syslog each time the created objects change:

```
Jan 22 11:24:35 totila clixon_backend[212183]: controller_edit_config:2728: Objects␣
↪created in actions-db: <services xmlns="http://clicon.org/controller" xmlns:nc=
↪"urn:ietf:params:xml:ns:netconf:base:1.0"><ssh-users xmlns="urn:example:test"><name>
↪test1</name><created nc:operation="merge"><path>/devices/device[name="openconfig1"]/
↪config/system/aaa/authentication/users/user[username="test1"]</path><path>/devices/
↪device[name="openconfig2"]/config/system/aaa/authentication/users/user[username="test1
↪"]</path></created></ssh-users></services>
```

5.6 5.6 Editing

Editing can be made by modifying services:

```
cli# set services test 2 eth* 1500
```

Editing changes the controller candidate, changes can be viewed with:

```
cli# show compare
     services {
+       test 2 {
+          name eth*;
+          mtu 1500;
+       }
     }
```

5.6.1 5.6.1 Editing devices

Device configurations can also be directly edited:

```
cli# set devices device example1 config interfaces interface eth0 mtu 1500
```

Show and editing commands can be made on multiple devices at once using "glob" patterns:

```
cli> show config xml devices device example* config interfaces interface eth0
example1:
<interface>
   <name>eth0</name>
   <mtu>1500</mtu>
</interface>
example2:
<interface>
   <name>eth0</name>
   <mtu>1500</mtu>
</interface>
```

Modifications using set, merge and delete can also be applied on multiple devices:

```
cli# set devices device example* config interfaces interface eth0 mtu 9600
cli#
```

5.7 5.7 Commits

This section describes *remote* commit, i.e., commit operations that have to do with modifying remote device configuration. See Section *devices* for how to make local commits for setting up device connections.

5.7.1 5.7.1 commit diff

Assuming a service has changed as shown in the previous secion, the *commit diff* command shows the result of running the service scripts modifying the device configs, but with no commits actually done:

```
cli# commit diff
     services {
+       test 2 {
+          name eth*;
+          add 1500;
+       }
     }
     devices {
        device example1 {
           config {
              interfaces {
                 interface eth0 {
-                   mtu 1400;
+                   mtu 1500;
                 }
              }
```

(continues on next page)

(continued from previous page)

```
            }
         }
         device example33 {
            config {
               interfaces {
                  interface eth3 {
-                    mtu 1400;
+                    mtu 1500;
                  }
               }
            }
         }
      }
```

5.7.2 5.7.2 Commit push

The changes can now be pushed and committed to the devices:

```
cli# commit push
```

If there are no services, changes will be pushed and committed without invoking any service handlers.

If the commit fails for any reason, the error is printed and the changes remain as prior to the commit call:

```
cli# commit push
Failed: device example1 validation failed
Failed: device example2 out-of-sync
```

A non-recoverable error that requires manual intervention is shown as:

```
cli# commit push
Non-recoverable error: device example2: remote peer disconnected
```

To validate the configuration on the remote devices, use the following command:

```
cli# validate push
```

If you want to rollback the current edits, use discard:

```
cli# discard
```

One can also choose to not push the changes to the remote devices:

```
cli# commit local
```

This is useful for setting up device connections. If a local commit is performed for remote device config, you need to make an explicit *push* as described in Section *Explicit push*.

5.7.3 5.7.3 Limitations

The following combinations result in an error when making a remote commit:

1) No devices are present. However, it is allowed if no remote validate/commit is made. You may want to dryrun service python code for example even if no devices are present.
2) Local device fields are changed. These may potentially effect the device connection and should be made using regular netconf local commit followed by rpc connection-change, as described in Section *devices*.
3) One of the devices is not in an OPEN state. Also in this case is it allowed if no remote valicate/commit is made, which means you can do local operations (like *commit diff*) even when devices are down.

Further, avoid doing BOTH local and remote edits simultaneously. The system detects local edits (according to (2) above) but if one instead uses local commit, the remote edits need to be explicitly pushed

Compare and check ==========– The "show compare" command shows the difference between candidate and running, ie not committed changes. A variant is the following that compares with the actual remote config:

```
cli> show devices <devices> diff
```

This is acheived by making a "transient" pull that does not replace the local device config.

Further, the following command checks whether devices are is out-of-sync:

```
cli> show devices <devices> check
Failed: device example2 is out-of-sync
```

Out-of-sync means that a change in the remote device config has been made, such as a manual edit, since the last "pull". You can resolve an out-of-sync state with the "pull" command.

5.8 5.8 Explicit push

There are also explicit sync commands that are implicitly made in *commit push*. Explicit pushes may be necessary if local commits are made (eg *commit local*) which needs an explicit push. Or if a new device has been off-line:

```
cli> push <devices>
```

Push the configuration to the devices, validate it and then revert:

```
cli> push <devices> validate
```

5.9 5.9 Templates

The controller has a simple template mechanism for applying configurations to several devices at once. The template mechanism uses variable substitution.

A limitation is that the template itself need to be entered as XML or JSON, CLI editing is not available.

Note: You need to enter the template as XML

Using of a template follows the following steps:

1) Add a template using the *load* command and commit it

2) Apply the template using variable binding on a set of devices

3) Commit the change

5.9.1 5.9.1 Limitations

Templates are added as raw XML. The reason is that YANG-binding is not known at the time of template creation. To know the YANG, the template needs to be bound to some specific YANG files, or specific devices.

Since it is raw XML, there is no type-checking and any diffs (based on YANG) is limited.

Note: Template XML is not type-checked and diffs are limited

5.9.2 5.9.2 Example

The following example first configures a template with the formal parameters *$NAME* and *$TYPE* using the load command to paste the template config directly:

```
> clixon_cli -f /usr/local/etc/clixon/controller.xml -m configure
olof@totila[/]# load merge xml
<config>
   <devices xmlns="http://clicon.org/controller">
      <template nc:operation="replace">
         <name>interfaces</name>
         <variables>
            <variable>
               <name>NAME</name>
            </variable>
            <variable>
               <name>TYPE</name>
            </variable>
         </variables>
         <config>
            <interfaces xmlns="http://openconfig.net/yang/interfaces">
               <interface>
                  <name>${NAME}</name>
                  <config>
                     <name>${NAME}</name>
                     <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">$
↪{TYPE}</type>
                  </config>
               </interface>
            </interfaces>
         </config>
      </template>
   </devices>
</config>
^D
olof@totila[/]# commit
olof@totila[/]#
```

Then, the template is applied: A ew *z* interface is created on all *openconfig* devices:

```
olof@totila[/]# apply template interfaces openconfig* variables NAME z TYPE ianaift:v35
olof@totila[/]# show compare
             openconfig-interfaces:interfaces {
+               interface z {
+                  config {
+                     name z;
+                     type ianaift:v35;
+                  }
+               }
             }
             openconfig-interfaces:interfaces {
+               interface z {
+                  config {
+                     name z;
+                     type ianaift:v35;
+                  }
+               }
             }
olof@totila[/]# commit
olof@totila[/]#
```

CHAPTER

SIX

6 YANG

6.1 6.1 Searching

6.1.1 6.1.1 Uniqueness

The controller YANGs rely on uniqueness of revisions. This means that even with schema mounts, all YANGs are part of the same search domain wrt revisions. That is, you may not have two different YANGs haveing the same revision.

6.1.2 6.1.2 Search path

Because of the uniqueness criterium, the controller uses the same search path for all YANGs. Typically the top-level search path is:

```
<CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
```

This includes all standard YANGs, clixon YANGs and controller YANGs.

Controller YANGs

The top-level of the controller-specific YANGs is typically */usr/local/share/clixon/controller*.

This can be changed with *configure –with-yang-installdir=DIR*, see Section *Installation*.

The controller YANG directory has two sub-directories with specific meanings:

- *main*. Main controller YANGs for the top-level. Note: only place YANGs here if you want them loaded to the top-level. Important: do not place device YANGs there, only controller YANGs.
- *mounts*. YANGs retreived from devices using RFC 6022 *get-schema* are written here. You can also add local cached YANGs here.

Note: Do not place device YANGs in the *main* directory

6.1.3 6.1.3 Device YANGs

There are two mechanisms to get YANGs from devices:

1. Dynamic RFC6022 get-schema
2. Locally defined

Dynamic

RFC6022 YANG Module for NETCONF Monitoring defines a protocol for retrieving YANG schemas. Clixon implements this as a main mechanism.

This is automatically invoked if the device advertises "urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring" in the hello protocol. The state-machine mechanism for this is described in Section *transactions*.

Local

You can also declare a module-set which is loaded unconditionally in a device, or device-profile. In the following example, openconfig is declared as locally loaded:

```
<devices xmlns="http://clicon.org/controller">
   <device-profile>
      <name>myprofile</name>
      <module-set>
         <module>
            <name>openconfig-interfaces</name>
            <namespace>http://openconfig.net/yang/interfaces</namespace>
         </module>
      </module-set>
   </device-profile>
</devices>
```

Note the following:

1. The locally defined openconfig YANG will searched for using the regular YANG search mechanism (using *CLICON_YANG_DIR*).
2. If the local YANG is not found, the (connect) transaction will fail.
3. Any imports declared in a locally defined YANG will also be loaded locally recursively
4. If the device also supports RFC 6022 get-schema, any further YANGs will be loaded from the device.

6.2 6.2 Structure

6.2.1 6.2.1 Clixon-controller

The clixon-controller YANG has the following structure:

```
module: clixon-controller
  +--rw processes
  |   +--rw services
```

(continues on next page)

(continued from previous page)

```
|     +--rw enabled              boolean
+--rw services
|   +--rw properties
+--rw devices
|   +--rw device-timeout         uint32
|   +--rw device-group* [name]
|   | +--rw name                 string
|   | +--rw description?         string
|   | +--rw device-group*        leafref
|   +--rw device-profile* [name]
|   | +--rw name                 string
|   | +--rw description?         string
|   | +--rw user?                string
|   | +--rw conn-type            connection-type
|   | +--rw ssh-stricthostkey    boolean
|   | +--rw yang-config?         yang-config
|   +--rw device* [name]
|     +--rw name                 string
|     +--rw enabled?             boolean
|     +--rw device-profile       leafref
|     +--rw description?         string
|     +--rw user?                string
|     +--rw conn-type            connection-type
|     +--rw ssh-stricthostkey    boolean
|     +--rw yang-config?         yang-config
|     +--rw device-type          string
|     +--rw addr                 string
|     +--ro conn-state           connection-state
|     +--ro conn-state-timestamp yang:date-and-time
|     +--ro capabilities
|     | +--ro capability*        string
|     +--ro sync-timestamp       yang:date-and-time
|     +--ro logmsg               string
|     +--rw config
+--ro transactions
    +--ro transaction* [tid]
      +--ro tid                  uint64
      +--ro state                transaction-state
      +--ro result               transaction-result
      +--ro description          string
      +--ro origin               string
      +--ro reason               string
      +--ro warning              string
      +--ro timestamp            yang:date-and-time
notifications:
  +---n services-commit
  |   +--ro tid                  uint64
  +---n controller-transaction
      +--ro tid                  uint64
rpcs:
    +--config-pull
    +--controller-commit
```

(continues on next page)

(continued from previous page)

```
      +--connection-change
      +--get-device-config
      +--transaction-error
      +--transaction-actions-done
      +--datastore-diff
      +--device-template-apply
```

6.2.2 6.2.2 Service augment

The services section contains user-defined services not provided by the controller. A user adds services definitions using YANG *augment*. For example:

```
import clixon-controller { prefix ctrl; }
augment "/ctrl:services" {
    list myservice {
        ...
```

6.2.3 6.2.3 Controller-config

The clixon-controller-config YANG extends the basic clixon-config with several fields. These have previously been described in Section *configuration*. The structure is as follows:

```
module: clixon-controller-config
  augment /cc:clixon-config
  +--rw CONTROLLER_ACTION_COMMAND
  +--rw CONTROLLER_PYAPI_MODULE_PATH
  +--rw CONTROLLER_PYAPI_MODULE_FILTER
  +--rw CONTROLLER_PYAPI_PIDFILE
  +--rw CONTROLLER_YANG_SCHEMA_MOUNT_DIR
```

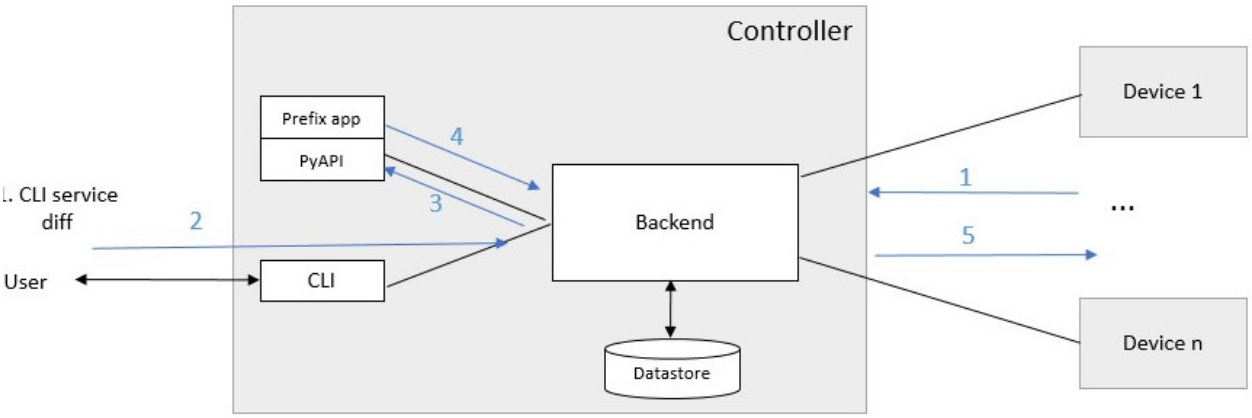
CHAPTER

SEVEN

7 TRANSACTIONS

A basis of controller operation is the use of transactions. Clixon itself has underlying candidate/running datastore transactions. The controller expands the transaction concept to span multiple devices. There are two such types of composite transactions:

1. *Device connect*: where devices are connected via NETCONF over ssh, key exchange, YANG retrieval and config pull
2. *Config push*: where a service is (optionally) edited, changed device config is pushed to remote devices via NETCONF.



7.1 7.1 Device connect

A *device connect* transaction starts in state *CLOSED* and if succesful stops in *OPEN*. there are multiple intermediate steps as follows (for each device):

1. An SSH session is created to the IP address of the device
2. An SSH login is made which requires:
   a) The device to have enabled a NETCONF ssh sub-system
   b) The public key of the controller to be installed on the device
   c) The public key of the device to be in the *known_hosts* file of the controller
3. A mutual NETCONF <*hello*> exchange
4. Get all YANG schema identifiers from the device using the ietf-netconf-monitoring schema.
5. For each YANG schema identifier, make a <*get-schema*> RPC call (unless already retrieved).

6. Get the full configuration of the device.

7.2 7.2 Config push

While a *device connect* operates on individual devices, the *config push* transaction operates on all devices. It starts in *OPEN* for all devices and ends in *OPEN* for all devices involved in the transaction:

1. The user edits a service definition and commits
2. The commit triggers PyAPI services code, which rewrites the device config
3. Alternatively, the user edits the device configuration manually
4. The updated device config is validated by the controller
5. The remote device candidate datastore is locked for exclusive access
6. The remote device is checked for updates, if it is out of sync, the transaction is aborted
7. The new config is pushed to the remote devices
8. The new config is validated on the remote devices
9. If validation succeeds on all remote devices, the new config is committed to all devices
10. If validation is not successful, or only a *push validate* was requested, the config is reverted on all remote devices.
11. The remote device candidate datastores are unlocked

After (9) above it is possible to add an extra step (compiler-option):

10. The new config is retreived from the device and is installed on the controller

Use the show transaction command to get details about transactions:

```
cli> show transaction
  <transaction>
     <tid>2</tid>
     <state>DONE</state>
     <result>FAILED</result>
     <description>pull</description>
     <origin>example1</origin>
     <reason>validation failed</reason>
     <timestamp>2023-03-27T18:41:59.031690Z</timestamp>
  </transaction>
```

7.2.1 7.2.1 Out-of-sync

In step (5) of the push algorithm described above, the remote device is checked for updates.

The controller employs a raw method for detecting this as follows:

1. Continuosly store the most recent device config on local storage. This is the "SYNCED" configuration, typically stored at */usr/local/var/controller*. This is either the most recent pull, or most recent push.
2. Get the complete configuration from the device as part of the transaction. This is the *TRANSIENT* configuration.
3. Compare the SYNCED and TRANSIENT configurations. If they differ, the device configuration has changed and the transaction is aborted.

A failed comparison is an indication that the device configuration has changed, and that therefore the push is unsafe since it may overwrite configuration entered by another party, such as a manual configuration of the device.

However, some devices rewrite fields automatically. Particularly in the case of a *push*, some devices themselves rewrite fields. Examples include encrypted or generated fdata, such as certs, keys, passwords or other data which for some reason are transformed at the time of the (push) commit.

Therefore, these fields cannot be used as a basis for equivalence and needs to be ignored in the out-of-sync comparison.

As a side note, an improved method than the raw algorithm described would be preferred, such as the device itself computing a hash value of its existing configuration.

7.2.2 7.2.2 Ignoring fields

The controller has a mechanism for ignoring device YANG fields by using a local file that augments the device YANG with an "ignore" extension.

For example, assume a "passwd" field should be ignored in a device YANG. First, add or extend a local YANG file:

```
module myext {
   ...
   namespace ""urn:example:ext";
   import device-yang {
      prefix dy;
   }
   import clixon-lib {
      prefix cl;
   }
   augment "/dy:configuration/dy:system/dy:passwd" {
      cl:ignore-compare;
   }
```

where the clixon-lib "ignore-compare" extension augments the passwd field in the original device YANG.

Then add it to a device or device-profile configuration:

```
device-profile my-device {
   ...
   module-set {
      module myext {
         namespace ""urn:example:ext";
      }
   }
   ...
}
```

When the device YANG is loaded, it will be augmented with the ignore extension, which the controller will use in its comparison algorithm.

CHAPTER

EIGHT

8 SERVICE API

The controller provides an *service API* which is a YANG-defined protocol for external action handlers, including the *PyAPI*.

The backend implements a tagging mechanism to keep track of what parts of the configuration tree were created by which services. In this way, reference counts are maintained so that objects can be removed in a correct way if multiple services create the same object.

There are some restrictions on the current service API:

1. Only a single action handler is supported, which means that a single action handler handles all services.
2. The algorithm is not hierarchical, that is, if there is a tag on a device object, tags on children are not considered
3. No persistence: if the backend is restarted, tags are lost.

8.1 8.1 Service instance

A service extends the controller yang as described in the YANG section section. For example, a service *ssh-users* may augment the original as follows:

```
augment "/ctrl:services" {
   list ssh-users {   // YANG list
      key group;      // Single key
      leaf group {
         type string;
      }
      list username {
         key name;
         leaf name{
            type string;
         }
         leaf ssh-key {
            type string;
         }
      }
   }
}
```

The service must be on the following form:

1. The top-level is a YANG list (eg *ssh-users* above)
2. The list has a single key (eg *group* above)

The rest of the augmented service can have any form (eg *list username* above).

Note: An augmented service must start with a YANG list with a single key

An example service XML for *ssh-users* is:

```
<services xmlns="http://clicon.org/controller">
  <ssh-users xmlns="urn:example:test">
     <group>ops</group>
     <username>
        <name>eric</name>
        <ssh-key>ssh-rsa AAA...</ssh-key>
     </username>
     <username>
        <name>alice</name>
        <ssh-key>ssh-rsa AAA...</ssh-key>
     </username>
  </ssh-users>
  <ssh-users xmlns="urn:example:test">
     <group>devs</group>
     <username>
        <name>kim</name>
        <ssh-key>ssh-rsa AAA...</ssh-key>
     </username>
     <username>
        <name>alice</name>
        <ssh-key>ssh-rsa AAA...</ssh-key>
     </username>
  </ssh-users>
</services>
```

The service protocol defines a service instances as:

```
<list>  |  <list>[<key>='<value>']
```

From the example YANG above, examples of service instances of *ssh-users* are:

```
ssh-users
ssh-users[group='ops']
ssh-users[group='devs']
```

where the first identifies all *ssh-users* instances and the other two identifies the specific instances given above

8.2 8.2 Device config

The service definition is input to changing the device config, where the actual change is made by Python code in the PyAPI.

A device configuration could be as follows (inspired by openconfig):

```
container users {
   description "Enclosing container list of local users";
   list user {
      key "username";
      description "List of local users on the system";
      leaf username {
          type string;
          description "Assigned username for this user";
      }
      leaf ssh-key {
          type string;
          description "SSH public key for the user (RSA or DSA)";
      }
   }
}
```

8.3 8.3 Tags

An action handler tags device configuration objects it creates with the name of the service instances using the *cl:creator* YANG extension. This is used to track which instance created an object and acts as a reference count when removing objects. An object may have several tags if it is created by more than one service instance.

In the following example, three device objects are tagged with service instances in one device, as follows:

Table 1: *Device A with service-instance tags*

Device object	Service-instance
eric	ssh-users[group='ops']
alice	ssh-users[group='devs']
kim	ssh-users[group='ops'], ssh-users[group='devs']

where device objects *eric* and *alice* are created by service instance *ops* (more precisely *ssh-users[group='ops']*) and *devs* respectively, and *kim* is created by both.

Suppose that service instance *ops* is deleted, then all device objects tagged with *ops* are deleted:

Table 2: *Device A after removal of ops*

Device object	Service-instance
alice	ssh-users[group='devs']
kim	ssh-users[group='devs']

Note that *kim* still remains since it was created by both ops and devs.

Note also that this example only considers a single device *A*. In reality there are many more devices.

8.4 8.4 Example python

An example PyAPI script takes the service ssh-users definition and creates users on the actual devices, for example:

```
for instance in root.services.users:
    for user in instance.username:
        username = ssh-users.name.cdata
        ssh_key = ssh-users.ssh_key.cdata
        for device in root.devices.device:
            new_user = Element("user",
                               attributes={
                                   "cl:creator": "users[group='ops']",
                                   "nc:operation": "merge",
                                   "xmlns:cl": "http://clicon.org/lib"})
            new_user.create("name", cdata=username)
            new_user.create("authentication")
            new_user.authentication.create("ssh-rsa")
            new_user.authentication.ssh_rsa.create("name", cdata=ssh_key)
            device.config.configuration.system.login.add(new_user)
```

8.5 8.5 Algorithm

The algorithm for managing device objects using tags is as follows. Consider a commit operation where some services have changed by adding, deleting or modifying service -instances:

1. The controller makes a diff of the candidate and running datastore and identifies all changed services-instances
2. For all changed service-instances S:

- For all device nodes D tagged with that service-instance tag:
  - If S is the only tag, delete D
  - Otherwise, delete the tag, but keep D

3. The controller sends a notification to the PYAPI including a list of modified service-instances S
4. The PyAPI creates device objects based on the service instances S, merges with the datastore and commits
5. The controller makes a diff between the modified datastore and running and pushes to the devices

The algorithm is stateless in the sense that the PyAPI recreates all objects of the modified service-instances. If a device object is not created, it is considered as deleted by the controller. Keeping track of deleted or changed service-instances is done only by the controller.

8.6 8.6 Protocol

The following diagram shows an overview of the action protocol:

```
Backend                            Action handler
   |                                    |
   + <--- <create-subscription> ---     +
   |                                    |
   +  --- <services-commit> --->        +
   |                                    |
   + <---   <edit-config>   ---         +
   |            ...                     |
   + <---   <edit-config>   ---         +
   |                                    |
   + <---  <trans-actions-done> ---     +
   |                                    |
   |          (wait)                    |
   +  --- <services-commit> --->        +
   |            ...                     |
```

where each message will be described in the following text.

8.6.1 8.6.1 Registration

An action handler registers subscriptions of service commits by using RFC 5277 notification streams:

```
<create-subscription>
   <stream>service-commit</stream>
</create-subscription>
```

8.6.2 8.6.2 Notification

Thereafter, controller notifications of type *service-commit* are sent from the backend to the action handler every time a *controller-commit* RPC is initiated with an *action* component. This is typically done when CLI commands *commit push*, *commit diff* and others are made.

An example of a *service-commit* notification is the following:

```
<services-commit>
   <tid>42</tid>
   <source>candidate</source>
   <target>actions</target>
   <service>ssh-users[group='ops']</service>
   <service>ssh-users[group='devs']</service>
</services-commit>
```

In the example above, the transaction-id is *42* and the services definitions are read from the *candidate* datastore. Updated device edits are written to the *actions* datastore.

The notification also informs the action server that two service instances have changed.

A special case is if *no* service-instance entries are present. If so, it means *all* services in the configuration should be re-applied.

8.6.3 8.6.3 Editing

In the following example, the PyAPI adds an object in the device configuration tagged with the service instance *ssh-users[group='ops']*:

```
<edit-config>
  <target><actions xmlns="http://clicon.org/controller"/></target>
  <config>
    <devices xmlns="http://clicon.org/controller">
      <device>
        <name>A</name>
        <config>
          <users xmlns="urn:example:users" xmlns:cl="http://clicon.org/lib" nc:operation=
↪"merge">
            <user cl:creator="ssh-users[group='ops']">
              <username>alice</username>>
              <ssh-key>ssh-rsa AAA...</ssh-key>
            </user>
        </users>
        </config>
      </device>
    </devices>
  </config>
</edit-config>
```

Note that the action handler needs to make a *get-config* to read the service definition. Further, there is no information about what changes to the services have been made. The idea is that the action handler reapplies a changed service and the backend sorts out any deletions using the tagging mechanism.

8.6.4 8.6.4 Finishing

When all modifications are done, the action handler issues a *transaction-actions-done* message to the backend:

```
<transaction-actions-done xmlns="http://clicon.org/controller">
  <tid>42</tid>
</transaction-actions-done>
```

After the *done* message has been sent, no further edits are made by the action handler, it waits for the next notification.

The backend, in turn, pushes the edits to the devices, or just shows the diff, or validates, depending on the original request parameters.

8.6.5 8.6.5 Error

The action handler can also issue an error to abort the transaction. For example:

```
<transaction-error>
  <tid>42</tid>
  <origin>pyapi</origin>
  <reason>No connection to external server</reason>
</transaction-error>
```

In this case, the backend terminates the transaction and signals an error to the originator, such as a CLI user.

Another source of error is if the backend does not receive a *done* message. In this case it will eventually timeout and also signal an error.

CHAPTER

NINE

9 PYTHON API

This section documents the Clixon Python API. The Python API is a stand-alone client which uses the internal Netconf protocol to the Clixon backend. The primary application is the clixon-server and its network services.

9.1 9.1 Overview

The Clixon Python API consist of two parts:

- The server (clixon_server.py).
- Service modules.

The server listens for events from the Clixon backend and run the modules when needed. All service logic are implemented in the modules and are described in detail below.

9.2 9.2 Overview

The Python API connect to Clixons internal socket and communicate over NETCONF. When started it registers for notifications of service commits and controller transactions.

Whenever a commit occurs, the Clixon Controller issues a notification the Python API listens to.

When the Python API receives a service commit it runs all the service modules which manipulates the configuration tree. When finished, the configuration is sent back to the Clixon backend.

In summary: 1. The user configures a service from the CLI. 2. The user commits the service. 3. Pyapi receives a <*services-commit*> notification. 4. Pyapi executes all service modules that may modify the configuration (device) tree. 5. For each modification, an <*edit-config*> message is sent to the backend. 6. When completed, pyapi sends <*transaction-actions-done*> to the backend. 7. If pyapi encounters an error, it aborts by sending <*transaction-error*> to the backend instead. 8. The new configuration is pushed to the devices by the backend.

9.3 9.3 Installation

9.3.1 9.3.1 Prerequisites

Installation of Cligen and Clixon is not covered in this section. The Clixon controller must be up and running before the Python API can be used.

It is expected that Python, Pip etc are installed on the system.

9.3.2 9.3.2 Installation

Python API is available on GitHub:

```
$ git clone https://github.com/clicon/clixon-pyapi.git
```

Once cloned, the requirementes are installed:

```
$ cd clixon-pyapi
$ pip3 install -r requirements.txt
```

And then the Clixon pyapi library is installed:

```
$ sudo python3 setup.py install
```

The server is installed manually, for example:

```
$ sudo cp clixon_server.py /usr/local/bin/
```

The install script *install.sh* performs the two steps above.

9.4 9.4 Usage

9.4.1 9.4.1 Command line options

The Python API server has the following command line options:

```
$ python3 clixon_server.py -h

 clixon_server.py -f<module1,module2> -s<path> -d -p<pidfile>
     -f       Clixon controller configuration file
     -m       Modules path
     -e       Comma separate list of modules to exclude
     -d       Enable verbose debug logging
     -s       Clixon socket path
     -p       Pidfile for Python server
     -F       Run in foreground
     -P       Prettyprint XML
     -l       <s|o> Log on (s)yslog, std(o)ut
     -h       This!
```

9.4.2 9.4.2 Logging and debugging

The server can be run in the foreground with debug flags:

```
clixon_server.py -F -d -P -f /usr/local/etc/controller.xml
```

9.4.3 9.4.3 Startup

Pyapi needs to know where the python code for the service model is located. This can be modified with the '-m' flag:

```
python3 ./clixon_server.py -f /usr/local/etc/controller.xml
```

which makes the server run in the background with minimal logging.

CHAPTER

TEN

10 SERVICE DEVELOPMENT

Service modules contains the actual code and logic which is used when modifying the configuration three for services.

The Python server looks for modules in the directory `/usr/local/clixcon/controller/modules` unless anything else is defined) and when a module is launched by the Python server the server call the setup method.

A minimal service module may look like this:

```
from clixon.clixon import rpc

SERVICE = "example"


@rpc()
def setup(root, log, **kwargs):
    log.info("I am a module")
```

10.1 10.1 Module installation

Clixon controller installs a utility named "clixon_controller_packages.sh" in "/usr/local/bin". This can be used to install packages.

```
$ clixon_controller_packages.sh -h
Usage: /usr/local/bin/clixon_controller_packages.sh [OPTIONS]
  -s Source path
  -m Clixon controller modules install path
  -y Clixon controller YANG install path
  -r Use with care: Reset Clixon controller modules and YANG paths
  -h help
```

The script copies Python code and module YANG files to the correct directories and take care of permissions etc.

The normal use case is to run the "clixon_controller_packages.sh" without the `-m` and `-y` arguments, the script installs modules and YANG in the default paths which is preferred.

10.2 10.2 Modules basics

The setup method take three parameters, root, log and kwargs.

- *Root* is the configuration three.
- *Log* is used for logging and is a reference to a Python logging object. The log parameter can be used to print log messages. If the server is running in the foreground the log messages can be seen in the terminal, otherwise they will be written to syslog.
- *kwargs* is a dict of optional arguments. kwargs can contain the argument "instance" which is the name of the current service instance that is being changed by the user.

There is also a variable named "SERVICE" that should have the same name as the service without revision.

```
from clixon.clixon import rpc

SERVICE = "example"

@rpc()
def setup(root, log, **kwargs):
    log.info("Informative log")
    log.error("Error log")
    log.debug("Debug log")
```

The root parameter is the configuration three received from the Clixon backend.

Contents of the root parameter can be written in XML format by using the dumps() method:

```
from clixon.clixon import rpc

SERVICE = "example"


@rpc()
def setup(root, log, **kwargs):
    log.debug(root.dumps())
```

10.3 10.3 Service attributes

When creating new nodes related to services it is important to append the proper attributes to the new node. The Clixon backend will keep track of which nodes belongs to which service using the attribute cl:creator where the value of cl:create is the service name.

Example:

```
from clixon.clixon import rpc

SERVICE = "example"


@rpc()
def setup(root, log, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    device.config.configuration.system.create("test", cdata="foo",
                      attributes={"cl:creator": "test-service"})
```

10.4 10.4 Python object tree

Manipulating the configuration tree is the central part of the service modules. For example, a service could be defined with the only purpose to change the hostname on devices.

In the Juniper CLI one would do something similar to this to configure the hostname:

```
admin@junos> configure
Entering configuration mode

[edit]
admin@junos# set system host-name foo-bar-baz

[edit]
admin@junos# commit
commit complete
```

However, in the Clixon CLI this behaviour can be modelled by using a service YANG models. For example, altering the hostname for a lot of devices could look as follows:

```
test@test> configure
test@test[/]# set services hostname test hostname foo-bar-baz
test@test[/]# commit
```

Clixon itself can not modify the configuration when the commit is issued, but this must be implemented using a service module.

```
from clixon.clixon import rpc


SERVICE = "example"

@rpc()
def setup(root, log, **kwargs):
    hostname = root.services.hostname.hostname

    for device in root.devices:
        device.config.configuration.system.host_name
```

When the service module above is executed Clixon automatically calls the setup method. The wrapper "rpc" takes care of fetching the configuration tree from Clixon and write the modified configuration back when the setup function returns.

The "root" object is modified and passed as a parameter to setup. It is parsed by the Python API and converted to a tree of Python objects.

One can also create new configurations. For example, the same example can be modified to create a new node named test:

```
from clixon.clixon import rpc


SERVICE = "example"

@rpc()
def setup(root, log, **kwargs):
    device.config.configuration.system.create("test", cdata="foo")
```

The code above would translate to an NETCONF/XML string which looks like this:

```
<device>
  <config>
    <configuration>
      <system>
        <test>
          foo
        </test>
      </system>
    </configuration>
  </config>
</device>
```

10.5 10.5 Object tree API

Clixon Python API contains a few methods to work with the configuration three.

10.5.1 10.5.1 Parsing

The most fundamental method is parse_string from parse.py, this method take any XML string and convert it to a tree of Python objects:

```
>>> from clixon.parser import parse_string
>>>
>>> xmlstr = "<xml><tags><tag>foo</tag></tags></xml>"
>>> root = parse_string(xmlstr)
>>> root.xml.tags.tag
foo
>>>
```

As seen in the example above an object (root) is returned from parse_string, root is a representation of the XML string xmlstr.

Something worth noting is that XML tags with '-' in them must be renamed. A tag named "foo-bar" will have the name "foo_bar" after being parsed since Python don't allow '-' in object names.

The original name is saved and when the object tree is converted back to XML the original name is be present:

```
>>> xmlstr = "<xml><tags><foo-bar>foo</foo-bar></tags></xml>"
>>> root = parse_string(xmlstr)
>>> root.xml.tags.foo_bar
```

(continues on next page)

(continued from previous page)

```
foo
>>> root.dumps()
'<xml><tags><foo-bar>foo</foo-bar></tags></xml>'
>>>
```

10.5.2 10.5.2 Creation

It is also possible to create the tree manually:

```
>>> from clixon.element import Element
>>>
>>> root = Element("root")
>>> root.create("xml")
>>> root.xml.create("tags")
>>> root.xml.tags.create("foo-bar", cdata="foo")
>>> root.dumps()
'<xml><tags><foo-bar>foo</foo-bar></tags></xml>'
>>>
```

10.5.3 10.5.3 Attributes

For any object it is possible to add attributes:

```
>>> root.xml.attributes = {"foo": "bar"}
>>> root.dumps()
'<xml foo="bar"><tags><foo-bar>foo</foo-bar></tags></xml>'
>>> root.xml.attributes["baz"] = "baz"
>>> root.dumps()
'<xml foo="bar" baz="baz"><tags><foo-bar>foo</foo-bar></tags></xml>'
>>>
```

The Python API is not aware of namespaces etc but the user must handle that.

10.5.4 10.5.4 Adding tags

A new tag can now be added to root and look at the generated XML using the method dumps():

```
>>> root.xml.create("foo", cdata="bar")
>>> root.dumps()
'<xml><tags><tag>foo</tag></tags><foo>bar</foo></xml>'
>>>
```

10.5.5 10.5.5 Renaming tags

If needed the tag can be renamed:

```
>>> root.xml.foo.rename("bar", "bar")
>>> root.dumps()
'<xml><tags><tag>foo</tag></tags><bar>bar</bar></xml>'
>>>
```

10.5.6 10.5.6 Removing tags

And remove the tag:

```
>>> root.xml.delete("bar")
>>> root.dumps()
'<xml><tags><tag>foo</tag></tags></xml>'
>>>
```

10.5.7 10.5.7 Altering CDATA

CDATA can be altered:

```
>>> root.xml.tags.tag
foo
>>> root.xml.tags.tag.cdata = "baz"
>>> root.xml.tags.tag
baz
>>> root.dumps()
'<xml><tags><tag>baz</tag></tags></xml>'
>>>
```

10.5.8 10.5.8 Iterate objects

We can also iterate over objects using tags:

```
>>> from clixon.parser import parse_string
>>>
>>> xmlstr = "<xml><tags><tag>foo</tag><tag>bar</tag><tag>baz</tag></tags></xml>"
>>> root = parse_string(xmlstr)
>>>
>>> for tag in root.xml.tags.tag:
...     print(tag)
...
foo
bar
baz
>>>
>>> xmlstr = "<xml><tags><tag>foo</tag></tags></xml>"
>>> root = parse_string(xmlstr)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> for tag in root.xml.tags.tag:
...     print(tag)
...
foo
```

As seen above, there is a an XML string with a list of tags that can be iterated.

10.5.9 10.5.9 Adding objects

Objects can also be added to the tree:

```
>>> root.dumps()
'<xml foo="bar" baz="baz"><tags><foo-bar>foo</foo-bar></tags></xml>'
>>> new_tag = Element("new-tag")
>>> new_tag.create("new-tag")
>>> root.xml.tags.add(new_tag)
>>> root.dumps()
'<xml foo="bar" baz="baz"><tags><foo-bar>foo</foo-bar><new-tag><new-tag/></new-tag></
↪tags></xml>'
>>>
```

The method add() adds the object to the tree and. The object must be an Element object.