Continuous Integration in CFMGR

W. David Fröhlingsdorf
Supervisors: Stefan Stancu and Adam Krajewski

CERN
CH-1211
Geneva 23
Switzerland

Cern Summer Student Programme — 19/6/17 to 8/9/17

Abstract

Cfmgr is a managing tool for network devices. At the moment there is no way to automatically check the working behaviour of the tool, meaning that a lot of effort is spend into manually testing the tool after an update. During my stay at CERN I developed a black-box testing framework for Cfmgr according to Continuous Integration practices and successfully deployed the framework using Jenkins and Docker. This report discusses in detail how the framework works and how it can be configured, and equally gives a broad problem description and outlines future work directions.

Contents

Chapter 1

Background

1.1 CFMGR

Cfmgr is a network configuration tool which has been developed by CERN to simplify and automate the configuration of network devices. Typically, network devices can be configured via a command line interface (accessible through Telnet) and SNMP. However, interfaces can vary greatly between vendors and firmware versions. Ergo, Cfmgr greatly simplifies the configuration of network devices in CERN's network by providing a uniform interface. Moreover, Cfmgr automatically fetches and modifies information from LANDB, CERN's database for all network devices and users.

Cfmgr is written in Perl and finding staff with the required skill and knowledge is becoming increasingly difficult. Maintenance is in particular complicated, especially for the core modules of the software. Substantial manual testing is needed to ensure that all specific procedures relying on the changed software still work as intended. Furthermore it is difficult and error-prone to identify all the components that might be affected by a change.

1.2 Continuous Integration

Continuous Integration (CI) is the process of automatically building and testing a code base each time that it has been changed. This approach has the huge advantage that accidentally introduced bugs can be detected right away and gives a degree of certainty that the changes have not introduced unwanted side-effects. Cfmgr could greatly benefit from continuous integration as it would make maintenance much easier.

1.2.1 Jenkins

One of the most popular tools for continuous integration is Jenkins. Jenkins is a server application written in Java. It can be used via command line or, more conveniently, via a web-based interface on TCP-port 8080. Typically Jenkins is used in combination with a version control system like git. Builds and tests can be triggered periodically, manually, each time a new commit or merge has occurred or any combination of these. It is necessary to create a web-hook on the version control repository manager (GitLab, GitHub etc.) to notify Jenkins each time a new commit or merge has been pushed to the remote repository.

Jenkins profits from a large amount of add-ons and is therefore extremely versatile. Furthermore, the pipeline feature allows the build stages to be defined in a Jenkinsfile which resides in the code base itself, meaning that any changes to the build stages will be version controlled too, making the project portable and self-contained.

1.2.2 Docker

Docker is a virtualisation software which can be best described as a light-weight virtual machine management system. The core entity of Docker are *images*. A Dockerfile specifies how an image is build by using a parent image and providing simple shell script steps. For example, the Dockerfile shown in Algorithm 1 could be used to create a simple Ubuntu image with Haskell installed on it.

Algorithm 1 Dockerfile to create an image with Haskell installed on Ubuntu

```
1: FROM ubuntu:14.04
2: RUN apt-get install haskell-platform
```

Such an image can then be used to create yet another image on top of it in the same manner. This makes docker images much more powerful compared to virtual machines as they can easily be reused and extended giving a very versatile environment. An image can be used to launch a docker *container*. A container is a running instance of a docker image. There can be arbitrary many containers of an image each independent from another, but extremely light-weight as they all share the same image.

I used Docker in combination with Jenkins in order to run all test cases inside a container. Jenkins first builds a base image from CentOS7 installing all required software for Cfmgr and the mock-emulator inside of it (see *mock_emulator/docker/machine*). Using this base image, Jenkins then adds the Cfmgr code base and configuration files into a deploy image (see *mock_emulator/docker/deploy*). Note that it is not possible to add parent files of the Dockerfile to an image. As a workaround Jenkins first copies the Dockerfile into the root of the repository as well as the secrets and configurations into the deploy directory. Jenkins deletes the created copies after a successful build of the image. However, should the build fail the copies remain in place until a successive Jenkins build is successful (see *Jenkinsfile*). Once the deploy image has been build, Jenkins starts a container of the image running the testsuite inside of it and copying over (from the container to the CI machine) the results once the testsuite has been finished.

1.2.3 Black-box testing

Black-box testing is a method of testing software without touching the source code itself. Even though it is restricted to a high-level inspection of the software, it offers some advantages over other testing paradigms. Firstly, no knowledge of the source code is required. This is particularly useful when the source code has a high complexity or is often changed. Secondly, black-box testing is not limited to a programming language. This means the software could be completely replaced as long as the high-level testing requirements are still fulfilled. These properties make black-box testing particularly interesting for testing Cfmgr.

Chapter 2

Objective

2.1 Aim

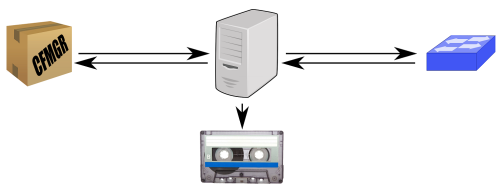
The aim of the project was to set up Jenkins such that it would trigger a command on Cfmgr to configure a switch and check that the switch has been configured as expected. However, this high-level problem description turned out to be not feasible in practice. In particular, the incorporation of a physical network device is highly problematic. It would require that the switch is always switched on and available for test cases. Jenkins would be required to log in to the switch and somehow verify that the switch has been configured correctly. Moreover, Jenkins would need to make sure that the switch is always in the same state before the test case is run. Finally, it would be difficult to test different firmware versions on the same switch as a physical device can only have a single firmware version installed at a time.

These reasons make it obvious that this approach is not practical and would make testing more complicated than necessary.

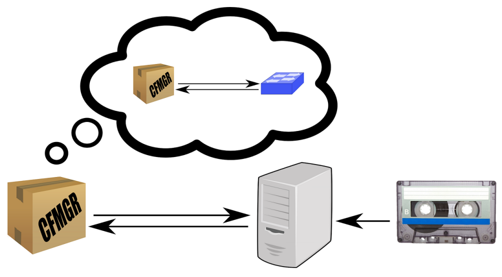
2.2 Plan of attack

As pointed out in Section 2.1 the initial project proposal is not easily implementable and would suffer from a limited test coverage. For this reason we followed a different approach to solve the problem.

Using a proxy between Cfmgr and the switch we are able to record communications on all used protocols (we call this *recording*, Figure 2.1a). In a second step, we can then use this recording and reply to requests from Cfmgr to the proxy without forwarding the requests (called *sequencing*, Figure 2.1b). From the point of view of



(a) Recording



(b) Sequencing

Figure 2.1: Plan of attack

Cfmgr there is no difference between recording and sequencing. However, should there be a request which does not correspond to our recording we can simply fail the test and stop replying to Cfmgr. Using a collection of recordings associated with commands we can easily set up Jenkins to test Cfmgr against this collection.

2.2.1 Code structure

Almost all relevant code for the mock-emulator resides inside the *mock_emulator* directory. All operations should be run using the Makefile which will automatically activate and deactivate the virtual environment as well as using authbind (allows non-root port bindings in privileged range). Typically the make will call a bash script in *mock_emulator/scripts* which handles the command. The command *make run_sequencer* can be used to start the sequencer and the commands *make run_recorder* and *make populate args=rec* (see Section 3.8) can be used to start a recording.

The directory *proxy_recorder* contains the recorder. *run_recorder.py* contains a handler class for recording and code to start the handler if the recorder is started with *make run_recorder*. The handler will load all recording configurations from *config.json*.

- `json-record` is the path where the recording will be saved
- `fail-record` is the path where a failed recording will be saved (useful for debugging)
- `snmp-addr` is the IP of the snmp proxy server (should almost always be 127.0.0.1)
- `snmp-port` is the SNMP port of the proxy server (should be 161)
- `tn-addr` is the IP of the telnet proxy server (should almost always be 127.0.0.1)
- `tn-port` is the telnet port of the proxy server (should be 23)
- `target-machine` is the IP address of the remote network device. In the future it should also be possible to use a URL here (see Section 4.3).

Recorder.py contains a class which is used to build up the recording file. By default new entries are appended to the *recording* field. However, the field can be changed if other data should be recorded. Note that all fields other than the *recording* field will be ignored by the sequencer. The handler sets up the recorder and starts all protocol proxies which can be found in the subdirectories (*mock_emulator/proxy_recorder/snmp_proxy* and *mock_emulator/proxy_recorder/telnet_proxy*).

The sequencer (in *mock_emulator/sequencer*) has been build to mirror the proxy recorder. *run_sequencer.py* contains a handler class for sequencing and code to start the handler if the sequencer is started with *make run_sequencer*. The handler will load all sequencer configurations from *config.json*.

- `json-record` is the path of the recording used for sequencing
- `snmp-addr` is the IP of the snmp server (should almost always be 127.0.0.1)
- `snmp-port` is the SNMP port of the server (should be 161)
- `tn-addr` is the IP of the telnet server (should almost always be 127.0.0.1)
- `tn-port` is the telnet port of the server (should be 23)
- `timeout` is the amount of time (in seconds) until the sequencer times out (failure because not all exchanges in the recording have been processed)

Sequencer.py handles the recording. In particular, it waits for requests from the protocols and replies according to the recording if the request matched the next exchange in the recording.

Chapter 3

Implementation

3.1 CFMGR Interaction

Typically the mock-emulator starts Cfmgr with a single command. However, often this is not enough and more subsequent user input is needed. For example, Cfmgr often needs confirmation (`[y]`) etc. In order to handle this requirement an interaction class has been build (*mock_emulator/cfmgr_expect/CfmgrExpect.py*). The functionality of this class is straightforward. Using a given interaction file which maps patterns to replies (each test case has its specific interaction patterns, see also Subsection 3.7), the run method triggers Cfmgr with the command and if any of the patterns occurs, the corresponding reply is sent. The method returns with a tuple containing the exit code and the output of the Cfmgr interaction.

Cfmgr expect can be configured with ***mock_emulator/cfmgr_expect/config.json***.

- `cfmgr-start` defines the location of *cfmgr.pl* relative to *mock_emulator*
- `timeout-seconds` defines in seconds how long Cfmgr should be given to complete or prompt for user input. This timeout should be reasonable long, as some operations require longer than a minute to complete, and should be at least as long as the sequencer timeout defined in *mock_emulator/sequencer/config.json*
- `cooldown-seconds` defines a wait time after the interaction with Cfmgr has finished. This cooldown is necessary in order to allow the recorder/sequencer to finish processing the final interactions in the pipeline. If no or a too short cooldown is set, recordings might become incomplete and/or the sequencer incorrectly reports an error.

3.2 Protocol Parameters

Occasionally the communication protocols (currently telnet and SNMP) are required to be configured differently from command to command. The ProtocolParams class in *mock_emulator/protocol_params/ProtocolParams.py* is responsible for this task using parameters for the configuration. All parameters are saved in a single object which is basically a sophisticated nested dictionary. There is no requirement about the structure of the dictionary, but by convention the first key layer should be the name of the protocol. Parameters are loaded from a json file which should be provided together with the recording. If no json file was provided or a requested parameter was not provided inside the json file, then the parameter's default value will be used instead. Therefore, each parameter request has to provide a default value which will be returned instead if the requested value can not be found.

If sensitive data needs to be provided to a protocol, it should not be provided directly in the json file. Instead a mapping should be provided which can be resolved during runtime. The class *mock_emulator/protocol_params/PreProcessParams.py* pre-processes all parameters and can therefore be used to execute this task. An example where this mapping is required are SNMP community strings as described in Subsection 3.3.2.

3.3 Sensitive Data

3.3.1 Passwords

The telnet recording should not capture usernames and passwords. Therefore the SensitiveChecker in *mock_emulator/mock_modules/SensitiveChecker.py* checks for replies from the network device which requests for such. The file *mock_emulator/mock_modules/telnet_expect_tokens.json* defines the patterns which signal that a username or password is about to be typed in. All subsequent telnet traffic originating from Cfmgr will be ignored until a newline or null-character (function `terminating_char` in *mock_emulator/mock_modules/stringifier.py*) has been received.

The equivalent happens in the sequencer. This means that while the sequencer is used any username/password combination is valid. A change in the authentication credentials will therefore have no effect on the recordings.

3.3.2 Community Strings

The pysnmp (which is used by the mock-emulator for SNMP traffic) library requires a community string to accept incoming requests. Similarly pysnmp requires a community string to connect to the remote network device. As those community strings should not be saved in the recording, a protocol parameter mapping is used instead (see Section 3.2). In particular, each device can either be unconfigured or configured. The according state is saved in the protocol parameter field `["snmp"]["community"]` as `public` and `private` respectively. The parameter pre processing class will call the community string dissolver in *mock_emulator/protocol_params/SNMPCommunityDissolver.py* which will use the device name and the parameter to load the read only and the read write community strings for the device. These read only and read write community strings will be set in the protocol parameters. Pysnmp will accept requests with either community strings but only the read-write community string is used to forward requests to the network device.

3.4 TFTP Snapshots

Often Cfmgr's behaviour depends on the content of the TFTP root as well as the cvs directory. To allow a reproducible test cases the populator automatically creates snapshots of the current tftp/cvs content and the testsuite restores the according snapshot for each test case respectively. The corresponding bash scripts are *mock_emulator/scripts/create_snapshot.sh* and *mock_emulator/scripts/apply_snapshot.sh*. Note that not all files, such as firmware files and skeletons, have to be saved as they remain the same throughout all snapshots. Therefore the snapshot archives can be kept light. By default, snapshots are saved in the *snapshots* directory next to the recording and have the same name as the corresponding recording (apart from the file extension `.tar.gz`, see also Subsection 3.7).

3.5 Simple test

The Makefile command *make simple_test* runs a single, user defined test case rather than the entire testsuite. This utility can be very useful for debugging. This single test case is defined by the file ***mock_emulator/tests/simple_test.json***.

- `rec_file` defines the location of the recording
- `cfmgr_script` defines the command to be given to Cfmgr
- `cfmgr_interaction` [optional]* defines the location of the interaction file with Cfmgr (see Section 3.1)
- `protocol_parameters_file` [optional]* defines the location of the protocol parameters (see also Section 3.2)
- `snapshot_file` [optional]* defines the location of the tftp snapshot archive (see Section 3.4)

*All optional fields have to be provided, but can be set to `null` if not needed.

3.6 Custom cases

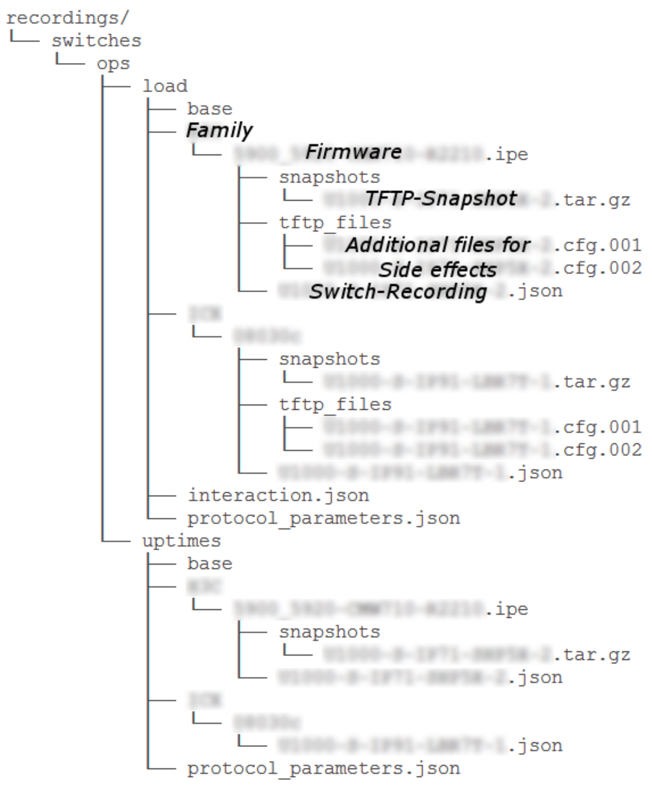
Custom cases, which are prepended to automatically configured test cases, can be defined in ***mock_emulator/tests/custom_cases.json***.

- `name` is the name of the test procedure
- `recording` is the location of the recording file
- `script` is the script to be given to Cfmgr
- `interaction` [optional]* defines the location of the Cfmgr interaction file (see Section 3.1)
- `protocol-parameters` [optional]* defines the location of the protocol parameter file (see Section 3.2)
- `snapshot` [optional]* defines the location of the tftp snapshot archive (see Section 3.4)

*Optional fields can be left out if not needed.

3.7 Customise automatically generated tests

The automatically generated test cases are based on recordings in the location *mock_emulator/tests/recordings* (see Figure 3.1). In particular the paths from the *recordings* directory to the location of the *base* files define the test commands for Cfmgr. The *base* file contains the location of the *interaction.json* and *protocol_parameters.json* files relative to the *base* file (one location per line). *interaction.json* defines the Cfmgr interaction (see Section 3.1) and *protocol_parameters.json* defines the protocol parameters (see Section 3.2) respectively. These files should normally be placed next to the *base* file to be not mistaken for a recording by the test case generator. Interaction and protocol parameter files are optional.



```
recordings/
└── switches
    └── ops
        ├── load
        │   ├── base
        │   ├── [redacted]
        │   │   └── [redacted].ipe
        │   │       ├── snapshots
        │   │       │   └── [redacted].tar.gz
        │   │       ├── tftp_files
        │   │       │   ├── [redacted].cfg.001
        │   │       │   └── [redacted].cfg.002
        │   │       └── [redacted].json
        │   ├── [redacted]
        │   │   └── [redacted]
        │   │       ├── snapshots
        │   │       │   └── [redacted].tar.gz
        │   │       ├── tftp_files
        │   │       │   ├── [redacted].cfg.001
        │   │       │   └── [redacted].cfg.002
        │   │       └── [redacted].json
        │   ├── interaction.json
        │   └── protocol_parameters.json
        └── uptimes
            ├── base
            ├── [redacted]
            │   └── [redacted].ipe
            │       ├── snapshots
            │       │   └── [redacted].tar.gz
            │       └── [redacted].json
            ├── [redacted]
            │   └── [redacted]
            │       └── [redacted].json
            └── protocol_parameters.json
```

Figure 3.1: Example - Recording tree

Recordings must have the exact device name as used in CSDB followed by the extension `.json`. They have to be placed in a directory tree below the *base* file which denotes the device family and firmware (for example *ICX/FW1/SWITCH-A.json*). The name of the firmware directory has to correspond exactly to the human readable output of Cfmgr when the `fware` command is run. If the device name begins with an underscore it indicates that the recording is empty and should be populated by the populator (see Section 3.8). All recordings beginning with an underscore are ignored when the testsuite is run.

TFTP snapshots (see Section 3.4) are placed in a directory *snapshots* next to the recording. Each snapshot is named in the same manner as the corresponding recording (`snapshots/SWITCH-A.tar.gz`). If for a recording no archive can be found, it is assumed that there is no snapshot for the recording.

Recording files can be amended. All fields apart from the `recording` field are completely ignored by the sequencer and have therefore no effect on the test cases. A recorded exchange (inside the `recording` field) can be extended with the `side-effect` field. The side effect must be a simple terminal command which will be executed in the location of the recording file before the reply is sent back to Cfmgr. A side effect might for example copy a file into the tftp root folder to simulate tftp traffic from the device to Cfmgr.

3.8 Populator

The mock-emulator profits from a tool called populator. The populator can automatically add and record new commands for the testsuite.

3.8.1 Command adder

More commands can be added to the testsuite by running *make populate args=add*. A prompt will ask for the command (input example: `switches ops fware`). Once all wanted commands have been added, an empty input will exit the prompt and all new commands will be created in *mock_emulator/tests/recordings*. In particular, the model in *mock_emulator/populator/model* will be copied to the corresponding positions. Before running the populate command, as described in Subsection 3.8.2, make sure to configure all *base* files of the newly added commands as needed (see Subsection 3.7).

3.8.2 Populate

The command *make populate args=rec* instructs the populator to check all recordings in *mock_emulator/tests/re cordings*. All empty recordings (that is, json files beginning with an underscore) will be first checked against the firmware on the target device as described in Subsection 3.8.3 and secondly automatically recorded. Moreover, the populator saves a tftp snapshot inside the neighbouring *snapshots* directory and saves the Cfmgr command to the recording in the field `command`. The `command` field is completely ignored by the testsuite and only exist for reference purposes.

3.8.3 Firmware checking

The populator checks automatically that the firmware version defined in the path of the recording matches the one of the machine. The class *mock_emulator/populator/FirmwareChecker.py* is handling this task. In particular, it triggers the corresponding firmware command on Cfmgr (for example for a switch the firmware checker would trigger `switches ops fware`) and records the communication in the recording field `fware`. The recorded firmware request communication is completely ignored by the testsuite and added to the recording file as a reference only. The final output of Cfmgr is handled as the firmware version. This output has to match the firmware in the recording's path and is also saved in the recording field `fware-human-readable`.

3.9 Request IDs

Each SNMP request PDU hast a request ID which the respond PDU has to contain (in other words, the request ID has to be echoed). The recorder and sequencer simply copy over the request ID to the response PDU.

3.10 Timestamps

Sometimes it is necessary to ignore a mismatch during sequencing. This is for example necessary if the exchange contains a timestamp. To ignore such an exchange a list of patterns in form of regular expressions has to be placed in the `ignore-patterns` field of the protocol parameters (for example `["telnet"]["ignore-patte rns"]`, see also Section 3.2). If any of these regular expressions match the `op` field of an exchange of the respective protocol, a mismatch in the exchange is ignored.

3.11 Channeliser

At times Cfmgr sends a message on one protocol and immediately sends a message on another protocol without waiting for a reply. Similarly, in a telnet connection traffic is often echoed or confirmed, but the sender is not waiting for a reply and keeps transmitting data instead. This can lead to very different sequences in the recording and needs to be avoided.

To handle this issue the class *mock_emulator/mock_modules/Channeliser.py* has been developed. The channeliser ensures that only one communication channel is open at a time. A communication channel is for example outgoing telnet traffic. Once no more traffic on the currently active channel is recorded, the channel times out and traffic on other channels is accepted. The channeliser uses layers of channels. For example, outgoing telnet traffic is a subchannel of the telnet channel. That means if the outgoing channel times out, the telnet channel becomes the active channel accepting any telnet traffic (ingoing or outgoing). If a subchannel is called, the active channel changes to the subchannel. If a superchannel is called, the active channel remains the current channel, but the timestamp is refreshed.

The channeliser uses a static stack, meaning that all channeliser objects work on the very same data. The channeliser is thread safe and uses a busy wait with sleep to wait for a channel to timeout. There is no obvious way to solve the timeout with a condition variable as a thread does not wait for another thread, but for a timeout to occur. More elegant solutions are possible, but the busy wait with sleep suits our problem just fine.

The channeliser can be configured with the configuration file ***mock_emulator/mock_modules/config_chann eliser.json***.

- `timeout` is the amount of time (in seconds) until the current channel is lifted (when there is no more traffic on the channel). In other words, if the channel times out, its parent channel will gain control.
- `spin-time` is the time (in seconds) that a thread which is waiting for a channel to become available is sleeping in each spin iteration. The sleep is necessary as otherwise a waiting thread could block the processor and therefore force the current channel to timeout.

3.12 Proxy class

The abstract class *mock_emulator/mock_modules/Proxy.py* (used by *mock_emulator/proxy_recorder/telnet_proxy /TelnetConveyor.py*) defines a generic server-proxy class for socket based byte streams which handles automatically channels (see Section 3.11). The class can be configured with ***mock_emulator/mock_modules/config_pro xy.json***. In particular the timeouts (server socket and connection socket) can be set. Having a reasonable short timeout is important as otherwise the proxy can not be stopped by the main thread.

3.13 OMAPI

OMAPI Perl is a Perl library, written by CERN, to map Perl code to the original OMAPI library written in C. Regrettably, OMAPI Perl has not been maintained for some time and depends on an old OMAPI version which is not available any more. Since Cfmgr requires OMAPI Perl in order to communicate with DHCP servers, OMAPI Perl had to be amended to work with the latest OMAPI version.

The current version of OMAPI Perl (`0.5.3`) requires a number of shared libraries, exported by OMAPI, in */usr/local/lib*. However, the latest version of OMAPI does not support all of these required libraries (libraries *libdns.a*, *libirs.a*, *libisc.a* and *libisccfg.a* are missing). Those missing libraries are still available inside OMAPI in the *dhcp-4.3.5/bind/lib* directory though. Equally, the include directory of OMAPI Perl must include *dhcp-4.3.5/bind/include*. In OMAPI Perl *Makefile.PL* and *CORE.xs* have been changed accordingly.

OMAPI and its bindings need to be compiled with the `-fPIC` flag in order to be able to use the libraries with OMAPI Perl (shared libraries). *dhcp-4.3.5/bind/Makefile.in* needs to be amended so that the `bindconfig` contains `CFLAGS=-fPIC`. Secondly, the OMAPI library needs to be configured with `./configure CFLAGS="-g -O2 -Wall -Werror -fno-strict-aliasing -fPIC"`.

Please study the Dockerfile in *mock_emulator/docker/machine* thoroughly for more information.

Chapter 4

Outcome

4.1 Recording structures

The *mock_emulator/tests/recordings* directory contains test cases in an intuitive and easily extendible hierarchy. As described in Section 3.8 the populator can be used to easily add and populate new recordings. Adding new firmware versions requires a little bit more work but can also easily be done by adding the necessary directories and empty recording files (starting with an underscore) to the *recordings* directory.

4.2 Recordings

Using the recorder and the sequencer we were able to test the six most common switch operations (`download`, `fware`, `init`, `load`, `stock` and `uptimes`) on two different brand switches in the lab. These test cases are automatically run by Jenkins in a Docker container on the CI VM each time new commits are pushed to the `ci-test` branch on GitLab (see also Subsection 1.2.2).

4.3 Future Work

Currently the mock-emulator uses */etc/hosts* to direct traffic from Cfmgr to the recorder/sequencer. The real IP address of the target machine has to be given to the proxy recorder in *mock_emulator/proxy_recorder/config.json* and is not automatically updated (in particular the populator, Section 3.8, uses the IP address given in the configuration). This is a significant limitation of the mock-emulator and can only be overcome by adding an `-ip` flag option to Cfmgr. The ideal position of the flag from the viewpoint of the mock-emulator would be between the command and the device name, because the position of the command (first) and the device name (last) are occasionally used in the code. Updating the mock-emulator to work with such an enhanced version of Cfmgr should not take too much work.

A further thinkable feature would be an automatic check of the behaviour of new firmware versions. In particular, recordings could be used to test if a switch with a newer firmware replies in the same way as with the firmware previously used for the recording.

Last but not least, the mock-emulator is implemented in Python and contains a strong separation of the logical and hardware layers. Should new Cfmgr modules be implemented in Python too, it would be possible to change the mock-emulator to a mock unit test, thus avoiding the network traffic altogether.