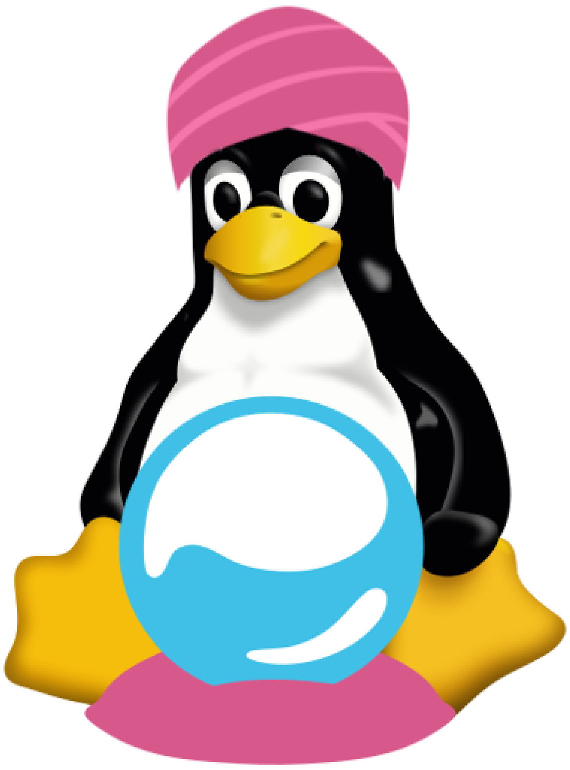
Projet TuxML
Manuel de reprise de code
ISTIC - Université de Rennes 1

Valentin PETIT Julien ROYON CHALENDARD
Cyril HAMON Paul SAFFRAY Michaël PICARD
Malo POLES Luis THOMAS Alexis BONNET

Encadrés par Mathieu ACHER



Lundi 22 Avril 2019

Ce rapport sera en Anglais, étant donné que son contenu sera aussi disponible sur le dépôt git du projet.

This report will be in English, since its content will also be available on the git project repository.

Table of Contents

1 Project overview

1.1 Project division

The project is separated in 3 part :

- outside the box : project's entries points and exit points
- inside the box : project docker image
- research and analysis script

Each of them have their dedicated part in this report.
This division can be seen at the root of the git repository, since the root content is four directory (compilation, docker_management, miscellaneous and tests), a python script (kernel_generator.py), a README.md and a LICENSE file.

1.2 Docker usage

1.2.1 What is Docker?

Quoting opensource.com :

> "Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.
>
> In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application."

1.2.2 Why are we using it?

In this project, it was mandatory to have a working environment which is controlled, stable and portable, while separating all the content of the project from the user environment. For this, we were in need of a virtual machine, but since a virtual machine is too cumbersome to use, we have chosen to use docker.

1.2.3 How it can be used, how do we use it?

Docker can be used as a command line application or with a Python API. Right now, we are using it as a command line application, but it could be changed to be used with the Python API later. Docker works with a set of images, who can be run inside a container. These images can be built with a Dockerfile or pulled from a distant Docker repository.

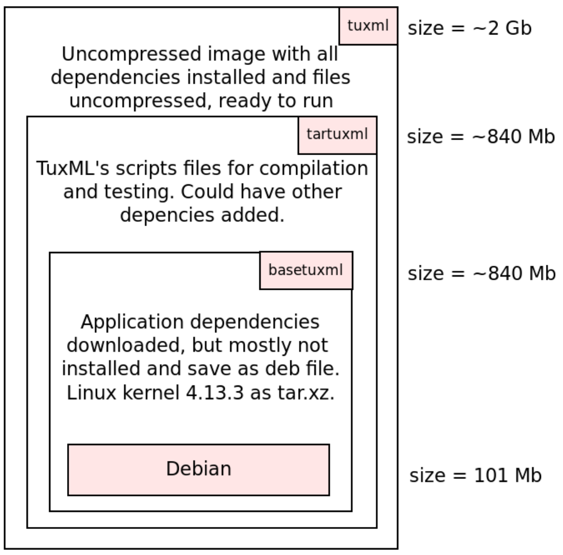
Each image can be differentiated by its name, its tag, its id or, when pulled from a repository, its digest, which is very useful for this project.

About the command, you can check the Docker documentation. If you don't do more than modifying the docker image content, all you need to know is how it is built and let our convenience script do the work.

1.2.4 Some word about our docker images

To begin, in the project the images are separated into 2 categories : the prod(uction) images and the dev(elopment) images. As their category says, the prod images are the latest stable images that can be used by the user, and they can be recognized by their tag which begins with prod. At the same time, the dev images are the latest image versions, but they can be unstable. As the prod image, you can recognized them with the dev tag.

In order to respond to some problematics (size, re-usability, maintainability, stability), a ready to run image of the project, is a layered structure of some images that you can see in the schema beside.


tuxml
size = ~2 Gb
Uncompressed image with all dependencies installed and files uncompressed, ready to run
tartuxml
size = ~840 Mb
TuxML's scripts files for compilation and testing. Could have other depencies added.
basetuxml
size = ~840 Mb
Application dependencies downloaded, but mostly not installed and save as deb file. Linux kernel 4.13.3 as tar.xz.
Debian
size = 101 Mb

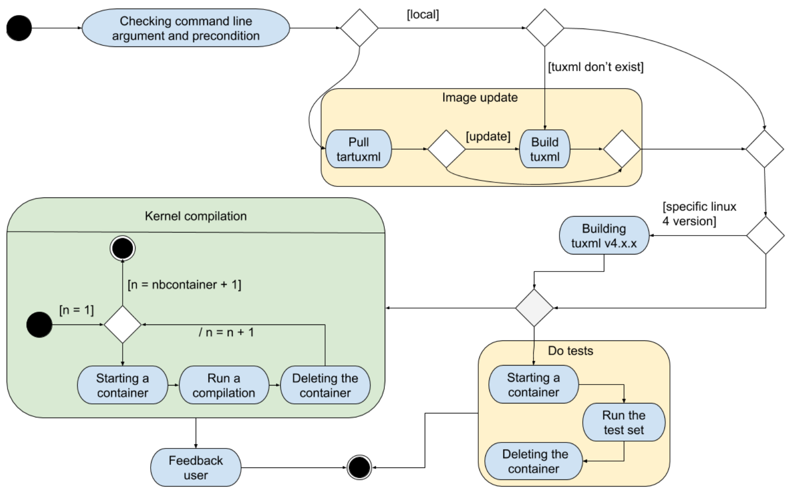
The tartuxml image is the one downloaded by the user, no matter the prod or dev tag. Afterward, each user will locally build the tuxml image with the corresponding tag, by decompressing the tartuxml image content. The decompression is simple and the same for every version of the image.

The user can afterward add an additional layer to replace the linux kernel with another version. In this case, the tag will be the old tag (dev or prod) follow by -v and the version. For example, with the 4.20.1 version and dev image, the tag will be dev-v4.20.1 .

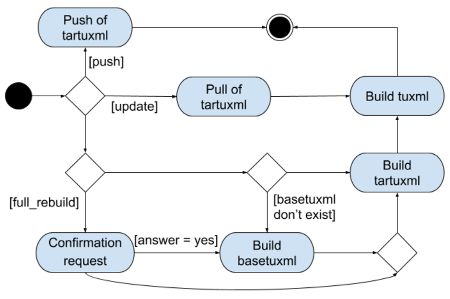
2 Outside the box

2.1 kernel_generator.py

This script is the principal entry point of the project, the standalone script which is downloaded by each user to run compilation and test. So, its goal is to manage everything for the user, from fetching the docker image to run a bunch of them and give a feedback. We will only provide the simplified activity diagram of this script, since its use is already presented in the User Manual and since it's enough to grasp how it works. For more, we invite you to read it and read some reports about it, that you can find in the project's wiki.



2.2 docker_image_tuxml.py



This script's goal is to provide an easier way to produce the docker image for the project while having the less possible knowledge of how to use docker or build an image.
So its workflow is pretty simple, as presented beside.
Thanks to that, its available command line arguments are also pretty self-explained, as you can see below.

```
$ ./docker_management/docker_image_tuxml.py --help
usage: docker_image_tuxml.py [-h] [-p] [-t TAG] [-d DEPENDENCIES] [-f]
                             [-l LOCATION] [-u]
```

```
optional arguments:
  -h, --help            show this help message and exit
  -p, --push            Push the image on the distant repository
  -t TAG, --tag TAG     Tag of the image you want to generate/build/push/pull.
                        Default to "dev"
  -d DEPENDENCIES, --dependencies DEPENDENCIES
                        Dependencies you want to add to your docker image when
                        you generate your dockerfile
  -f, --full_rebuild    Force the rebuild of the core system image, which is
                        not needed in most of the case.
  -l LOCATION, --location LOCATION
                        Where you want to create your directory to
                        generate/build. Default is current
  -u, --update          Download the image from the repository
```

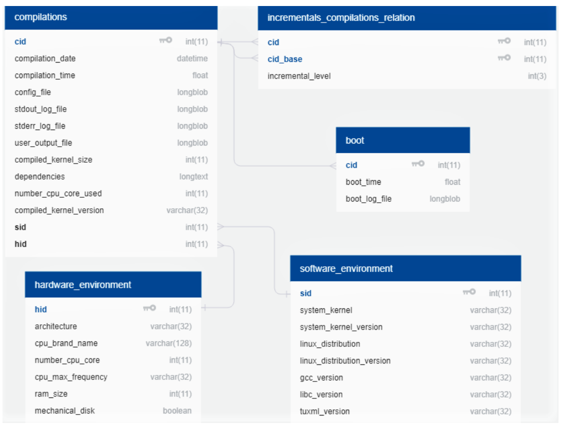
When building a new functionality, the task list is :

1. Modify Dockerfile content or embedded python file.
2. Build the image by running : $ ./docker_image_tuxml.py
3. Check if the functionnality is working without impacting others.
4. If yes, to push the image, run : $ ./docker_image_tuxml.py -p

2.3 Database

The database goal is to collect all the results of the compilations, and make them accessible to the user. This database design is a new one, which exists alongside the old one, and which emphasizes on size, re-usability and extensibility.

For more, check our report about the database in the project's wiki.



3 Inside the box

3.1 Docker image modularity

Along with the command line argument given with kernel_generator.py, the image has its own modularity :

- settings.py file, which contains practically every specific hard coded values used in the project. This file is located inside the compilation directory, which will be presented in the next section.
- kernel_version.txt file, which provides the version value of the kernel that the image will compile. The path to the uncompressed kernel directory is calculated according to this file content.
- dependencies.txt file, which gives the basic installed dependencies inside the image. Its content is loaded when starting a compilation.
- function and constructor parameters are at the most basic level, which permits a lot of modularity and reusability in order to reuse function or object for future goal.

3.2 Image content : compilation

The compilation directory is a fine group of scripts who work together to achieve the same goal : compile kernel and retrieve data on it. So, each script responds to a need or a goal :

- main.py : this is the entry point script, the one called outside the compiler to run the image. Its goal is to pull everything together and make it work as a whole while being simple enough to be understood without looking at every others scripts.
- logger.py : this script provides a logger object, which goal is to manage every single output of data inside the project, in order to produce the logs files and provide to the user some information on the execution progress.
- package_manager.py : this script provides an object whose goal is to manage every single system package needed to compile the image. At building, it contains all the installed packages when building the docker image.
- environment.py : this script provides utility functions to retrieve data about the environment. You will retrieve a dictionary containing all the hardware and software data thanks to it.

- configuration.py : this script is here to provide utility functions to list all the specific attributes of this run. They can be retrieved from environment or passed and analyzed from function parameter (retrieved from the main.py command line argument, for example).
- compiler.py : this script provides an object whose goal is to wrap every function related to the compilation of the kernel, from the config generation to the retrieving of the result.
- boot_checker.py : as the compiler script, this script provides an object to wrap every function related to the boot of a compiled kernel.
- database_management.py : this script provides utility functions to make communication with the database easier.

There are also some settings files : the settings.py file presented before, along with the tuxml.config and x64.config files, which serve as a base to make (respectively) a random or a tiny linux config file to compile.

3.3 Image content : test

Before even going to write some tests and think about them, we have to understand one important thing : in this project, we can't test every single part of the project, as we can't compile every single possible linux configuration.

So, for example, we can test every way of generating a config file, but we can't try to compile them, neither boot the compiled kernel that can be build. Also, you have to take care about concurrent access and system available space when testing.

Keeping this in mind, the tests directory contains scripts which are closely related to a script present in the compilation directory. In fact, their name is the tested script file, preceded by "test_". For, example we have a "test_environment.py".

So, we have scripts, that are completed or should be, which test the logger, the environment, the configuration, the compiler and the boot_checker scripts, while we can't test the database_management script, nor the package_manager script (in fact, we could check if the creation of the package manager object is successful or not).

4 Miscellaneous : others things about the project

In this part, we will present some of the other subjects that you could study outside of the actual development of the image. If some file code is used, it will be found inside the miscellaneous directory.

4.1 Old files

Some of the files are leftover of the old team work, which are left to provide information or code about old behaviour or research done.

4.2 Utilities script

Some utility scripts are present, to help us while developping.

- refactoring_test.py : this script goal is to fetch some old result from the old database, recompile the selected config and compare the result. It made us able to automate the test of the refactoring while populating the new database.
- log_decoder.py : this script goal is to make easier for a developer to decode a longblob file extracted from the database.

4.3 Study of Linux options and config file

Every script of this part is located in the specialconfig directory.

4.3.1 Randconfig

In order to fill the data set in the database, we need to provide a lot of different config files. Since, we can't create each config by hand, we use the randconfig option of the linux kernel Makefile. This method will set each option of a config file randomly, except the one that we preset. In other words, it's a random autocompletion tool.

The kconfig_checker.py was heavily used to refine our preset and check the good behaviour of randconfig.

4.3.2 Kconfiglib

During the project, a very large number of configurations were added to the database to analyze them and allow our Machine Learning algorithm to learn the behavior of these configurations. We needed a tool that would allow us to manage Kconfig, the configuration file or generate configurations (mainly a randconfig) and maybe even assist the configuration process or the extraction of dependencies/options information.

We think that Kconfiglib could allow us to manage and master Kconfig and the configuration files. We first used Kconfiglib to know the real number of options present in a linux kernel for all supported architectures and see their evolution according to the updates of the linux kernel (v4-0-1 to v4-20-1). In a second step, we tried to find out what was the purpose of certain options in order to know their importance and the impact they could have on a configuration.

We created 2 scripts using Kconfiglib, the documention of the use of Kconfiglib and scripts is here miscellaneous/specialconfig/Kconfiglib/README.md.

count_options.py :

The script miscellaneous/special_config/Kconfiglib/count_options.py is used to count the options of a specific kernel version. Kconfiglib upon execution generates a tree of the options in the kconfig files, the script then goes through every node of this tree and checks if the node is an option. This script was heavily used to create several histograms that represent the evolution of kernel's options according to the architecture.

get_option_prompt.py :

The script miscellaneous/special_config/Kconfiglib/get_option_prompt.py is used to prompt on the standard output the help section of an option of a kconfig file. The option should be written inside the script for now, in uppercase. As this script was written at the end of the project it can be greatly improved and might be a good entry point to grasp how kconfiglib and kconfig files work.

4.4 Incremental compilation

This is a whole field of research, to see the impact of compiling a slightly different Linux config file with the old leftover of previous compilation. The study measures differences in compilation time and kernel usability, while providing piece of data on each option's effect on Linux kernel compilation. Note that by kernel usability, we speak about the capacity of the kernel to run, since it looks like an incremental compilation sometimes break the kernel and made it unable to be run, while it would have been possible from scratch.

4.5 Going further

In order to go further, we strongly advise to read the whole project wiki to have a better grasp on all the project entity.