OULUN AMMATTIKORKEAKOULU

Mika Järvi

PORTING TEST ENVIRONMENT INTO PC SIMULATION

PORTING TEST ENVIRONMENT INTO PC SIMULATION

Mika Järvi
Bachelor's Thesis
Spring 2016
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author: Mika Järvi
Title of thesis: Porting Test Environment into Simulated Environment
Supervisor: Heikki Mattila
Term and year of completion: Spring 2016 Pages: 25

The aim of this Bachelor's thesis was to enable testing of one individual part, Layer2 application of base station software without a real target hardware. It was commissioned by Nokia. In the beginning there were existing test suites and a test environment, but to run tests one needed to queue up a test line whose amount is very limited.

The Layer2 application and support libraries were compiled and linked as executable for a traditional PC computer. The test material was modified to start the Layer2 application and run the tests directly at the development environment.

There were no actual impediments on completing the task. As a result, nearly 80% of the shock test suite are passing also when ran without the base station.

Keywords: Testing, Simulated, LTE

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tekniikan yksikkö, tietotekniikka

Tekijä: Mika Järvi
Opinnäytetyön nimi: Testijärjestelmän siirtäminen simuloituun ympäristöön
Työn ohjaaja: Heikki Mattila
Työn valmistumislukukausi ja –vuosi: kevät 2016 Sivumäärä: 25

Työn tavoitteena oli mahdollistaa tukiasemaohjelmiston Layer2 sovelluksen, testaaminen ilman todellista tukiasemaa. Työn tilasi Nokia. Lähtötilanteessa testit ja testiympäristö olivat valmiina mutta testien suorittamiseksi jonotettiin testipaikka joiden määrä on rajallinen.

Layer2 ja sen tarvitsemat oheiskirjastot käännettiin sovellukseksi perinteiseen PC tietokoneeseen. Testimateriaali muokattiin käynnistämään Layer2 sovellus ja suorittamaan halutut testit suoraan kehitysympäristössä.

Varsinaisia esteitä työn valmistumiselle ei ollut. Lopputuloksena 80% ”iskutesti” setistä voidaan suorittaa onnistuneesti myös ilman tukiasemaa.

Asiasanat: Testaus, Simulaatio, LTE

PREFACE

This thesis work was carried out within the autumn of 2015 and spring of 2016 in the Information Technology department at Oulu University of Applied Sciences.

First of all, I want to thank Heikki Mattila for supervising and Riitta Rontu for supporting me through all this time.

Also, I would like to thank Nokia for this task and making this all possible. Thanks to Tero Manninen for the company side support and supervision. Thanks to all colleagues in the LTE/Layer2 teams, and especially to Jari Helaakoski who solved the trickiest problems and saved the day so many times.

Oulu, 24.5.2016
Mika Järvi

TABLE OF CONTENTS

GLOSSARY

BTS	Base Transceiver Station
eNB	Evolved NodeB, LTE Base station
EU	Execution Unit
LTE	Long Term Evolution to improve wireless broadband
RLC	Radio Link Control protocol
SCT	System Component Testing
SUT	System Under Test
TDD	Test Driven Development
TTI	Transmit Time Interval, 1ms in LTE
UE	User equipment, mobile device
VoIP	Voice Over IP network

1 INTRODUCTION

In software development a feedback cycle is one key to productivity. To maintain a source code repository of a large software project in a good shape, it requires constant care, i.e. care in terms of renaming functions, rearranging variables, finding and eliminating duplication, restructuring the design - refactoring.

A unit test suite, which is fast and at least in as good shape as the code, is also a necessity. Unit tests alone are not enough to give the confidence before the commit.

When making a modification, a small change, which feels even a bit superfluous because “That change does not Change anything”, one must be 100% sure that it does not have any harmful effect which would spoil the whole good intention and turn it into disaster. To be fully sure is not possible. The closest thing would be to run all the possible tests that exist, but still the coverage typically has holes and after running all the possible tests, usually a lot of time has passed and the commit will involve rebasing, which in turn will water down the earlier tests.

A suite of effective acceptance tests to cover most of the usual breakages can offer a decent middle ground to reach enough confidence to make the commit happen. Therefore the good intention and feeling of being able to make the code better will overcome the slight uncertainty and occasional disasters.

Now, if running acceptance tests involves queuing a test line for several minutes and then waiting for a software to update for another several minutes, the suite is not fast enough. Moving from a real hardware to a simulated environment could gain some speed-up here.

2 TESTING EMBEDDED SOFTWARE

2.1 Testing aspects

Testing is an essential part of a software project. It often tends to require too much time yet providing a very little pay back for the investment. This applies especially when testing embedded software in a real target environment which suffers from delays, hardware related limitations and a high cost per test environment.

On the other hand, mistakes during the design and development are natural and unavoidable. Therefore, the best thing to do is to minimize the delay from introducing the error till it is found. It helps a developer to understand why the test failed because the modification since the last successful run is not too big. It also helps understanding how to make it right, and possibly saves a lot of unnecessary work, which would have been based on a wrong assumption. [1] In the book Test Driven Development for Embedded C, James Grenning writes about this in the context of unit testing but the same applies also to other levels of testing.

When testing larger entities, the amount of involved people typically increases. One reason is that the testing and development is divided to different persons. Another reason is that the testing is just so slow and testing resources are limited so that one test cycle must contain commits from several developers. This communication will add a new level of wasted time on communication and figuring out what was the cause of the failure. The more commits are involved in the change set and the more time passes from introducing the actual bug, the harder it will be to find out whose modification broke the test, what is wrong with it and how to fix it.

2.1.1 Design & Implement with unit tests

It is a known fact that the code is written once but read, modified and maintained for many years. If this maintenance happens without constant refactoring the code tends to grow ugly, simply because adding new lines of code without

touching the existing ones might be possible but typically leads to a cascading complexity. The fear of touching the existing parts of code tells about a fear of breaking something, not being able to easily verify “Does the code still do what it used to do”.

When a developer can test drive the implementation while working, the code can start to grow naturally into a “good shape” because all the expected behaviors and assumptions are recorded as small tests. In the first place, Test Driven Development moves the focus from implementing the function (solving the problem) on the interface and puts pressure on implementing testable and usable interfaces. Also, writing a call to the function before it even exists usually helps a developer to come up with a more descriptive name for the method itself and the arguments it takes. Having then a failing unit test in place, can help to find the simplest possible solution instead of over engineering something unnecessary complicated. It will also verify that new and modified code lines really do what the developer intended them to do. In the end, these tests act as a safety net for refactoring.

Unit tests are also important to fill in the gaps that higher tests cannot reach, for example, a network, a memory or other error conditions, which are hard to produce. Or for example, timing related occasions which are controllable only with fake collaborators.

2.1.2 Testing the smaller functionalities alone

In large and complicated systems, it is often reasonable to also test parts of the system alone to verify that each component of the system acts as specified. When tested on isolation, it is easier to understand the failure than when one part of a larger construction is causing the whole system to fail.

These tests can have a significant role on driving the architecture and design of the code towards clear and testable interfaces. When done right, these tests will assure that the internal interfaces in between the components are used as they are meant to be used and they do work as expected when used correctly. With-

out these tests the amount of combinations to be tested in a higher level will explode [2].

2.1.3 Visualize the big picture in the beginning

When moving on to a larger scale, the focus moves from small code lines to deliverable features. At the same time, testing moves towards the interfaces of the final product. From a software developer / a development team point of view, the earlier these “customer facing tests” can be ran, the more input they can get to their code design. In optimal situation the first few acceptance tests are implemented and ran already before starting the implementation, first of all to verify that the requested feature is not yet there. While implementing the feature, these tests serve as a feedback channel providing “what to do next” -ideas for the developers. In the end they verify that the feature is ready.

These early implemented failing tests can also act as a communication channel between developers and a customer to clearly state what the feature is about and what the actions on successful and unsuccessful use cases are [3].

2.2 Off target, why, how, alternatives

There are roughly two alternatives to run an application or parts of it without the real hardware, either to simulate the hardware or to simulate the Software API.

Both will decrease the startup time because there is no need to deploy the new software into the real target hardware and then wait for the restart to complete. On execution time it can be the opposite as the embedded real time processors are usually relatively fast, at least when comparing to the simulated hardware.

2.2.1 Hardware simulation

A cross compiled application can be ran on a development host, for example with a hardware simulator, which simulates the whole target processor, all used hardware interfaces and middleware (Figure 1). It makes it possible to also verify cross compilation and linking steps to a hardware as early as possible, offering a transition to the real hardware, which might not even be available in the beginning.

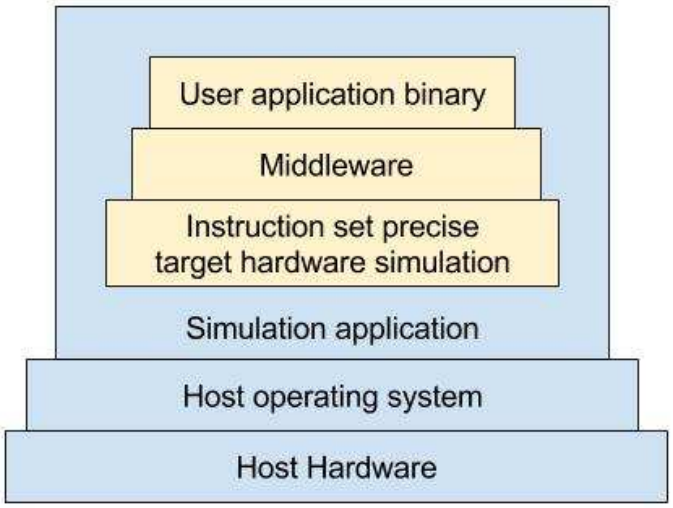

FIGURE 1. Hardware simulation

Running on a simulated hardware offers lots of possibilities such as testing easily with different hardware variants without physically really having all the real hardware variants. It can also provide a mechanism to reproduce a deterministic behavior on each test run as all the interrupts and task-switches will happen in a same order each time. A simulated environment also offers good possibilities for a measuring and profiling performance. From the debugging point of view, simulation is easier to freeze inspect step by step. [4]

As a downside, implementing and maintaining an interpreter, which is able to reliably simulate an instruction set of a target processor without any third party libraries or applications, is invariably not an option. These environments usually require engaging to some hardware vendor specific simulator or purchasing an expensive simulation solution, and when in action, these environments tend to eat lot of time, CPU processing power and memory.

2.2.2 Software API simulation (compile for the native)

Another approach would be to run an application as a native application on the development host (Figure 2). All the application sources must be built to an executable for the native host instruction set. This compilation involves an operating system and hardware API headers and declarations for services that are really available only in the target environment. Missing interfaces have to be implemented as “stubs”, “mocks” or “test doubles”, simple implementations of

the real services just enough to satisfy the linker when building the executable. Test doubles can contain triggers or observation mechanisms to serve different testing purposes. In some cases test doubles can even contain larger verifications, which would not be possible on a real target.

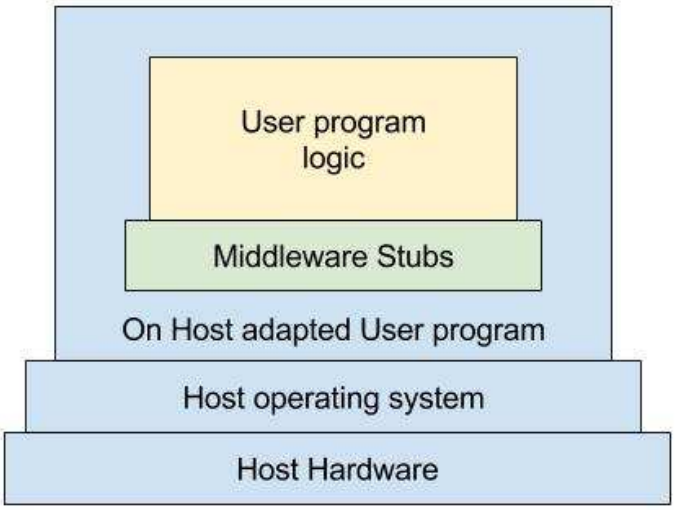

FIGURE 2. Native application with simulated API

A native compiler is usually fast compared to cross compilers provided by hardware vendors, like Texas Instruments, which often put a lot of time and effort in optimizing code for some specific hardware. Using generic and popular environments will open a wide range of native tools, e.g. for debugging or detecting runtime errors on memory access.

3 THE BASE STATION

3.1 Nokia Evolved NodeB

3GPP defines a base station as follows:

> A base station is a network element in radio access network responsible for radio transmission and reception in one or more cells to or from the user equipment. A base station can have an integrated antenna or be connected to an antenna by feeder cables. In UTRAN it terminates the Iub interface towards the RNC. In GERAN it terminates the Abis interface towards the BSC. [5]

Now as being an Evolved 4G Base Station, it also contains parts of controlling functionalities, which have traditionally been part of an RNC network element. This makes the LTE network architecture simpler and its response times faster.

The LTE/Layer2 system component is a group of one of the design blocks in the eNB architecture (Figure 3). It provides Medium Access Control (MAC), Radio Link Control (RLC) and Packet Data Convergence Protocol (PDCP) functionalities. [6]

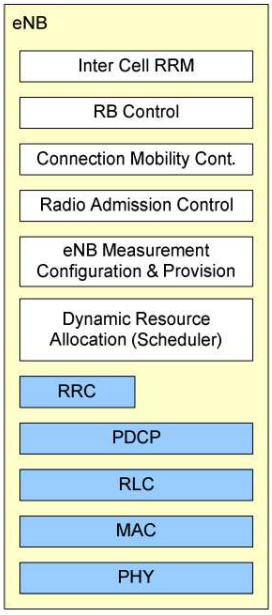

FIGURE 3. eNB high level architecture [6]

3.2 Layer2 in action

In practice LTE/Layer2 has two separated sides, sending a dataflow to the user equipment and receiving from user equipment (Figure 4). Both of these implement one peer of PDCP, RLC and MAC protocols.

On down link, sending to UE, side, once per every 1ms Transmit Time Interval, a small group of active users have been selected by the packet scheduler based on their priority, quality of service and other parameters. For each of those users a transport block will be created and filled with a given amount of bytes from given radio bearers.

Closest to the IP packet data there is the PDCP protocol which takes care of compressing an IP header which in turn is considerably large compared to, e.g. a streaming VoIP data payload. This protocol also removes possible duplicated packages and discards too old packages.

The next protocol is RLC, which then carries the PDCP Header and data based on the bearer specific mode which can be:

- Transparent, no segmentation, no delivery guarantees.
- Unacknowledged, segmentation and reassembly, no delivery guarantees
- Acknowledged, segmentation and reassembly, reliable delivery

Finally, the MAC protocol takes this RLC header, PDCP header and IP data as one of logical channels and multiplexes these logical channels into one transport block.

The MAC layer also implements a hybrid automatic repeat request (HARQ), which significantly improves the throughput on LTE. With this low level feature, a peer is able to get retransmissions of erroneous received transport blocks faster than with normal ARQ procedure that involves a more complicated RLC protocol re-segmentation.

A up link direction, receiving from the UE, works like a downlink but vice versa. It receives a transport block from each transmitting user. Based on the included header information, it then joins received pieces of data into a radio bearer streams. It also reports reception failures and requests retransmissions of erroneous transport blocks of the user equipment peer.

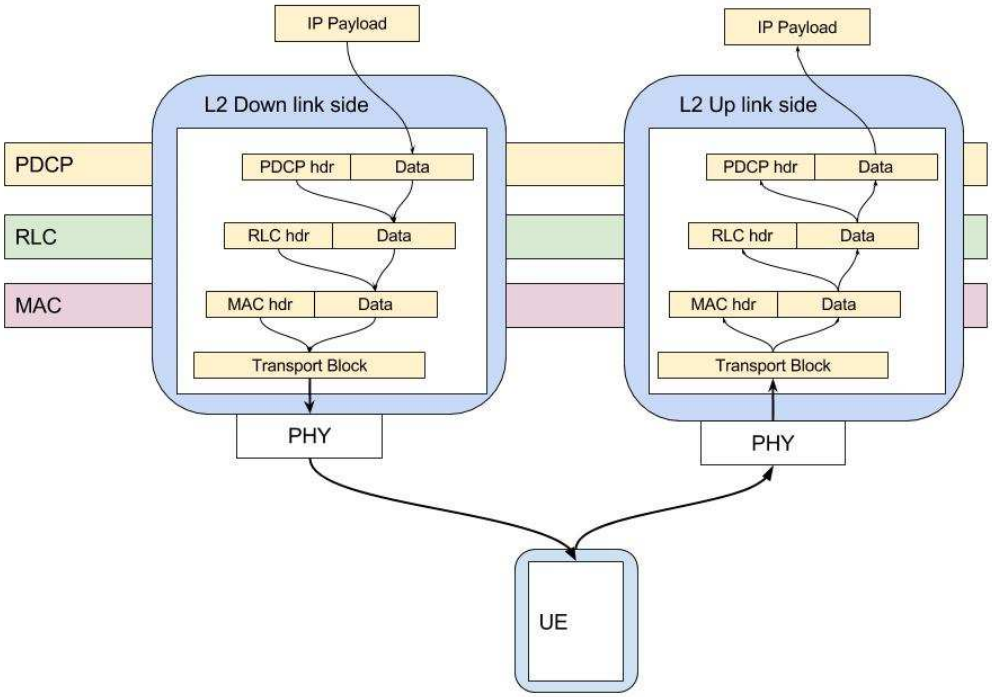

FIGURE 4. Simplified Layer2 dataflow

3.3 System Component Tests

System Component Tests are the first tests in which Layer 2 works as one fully integrated application. These test cases communicate with Layer2 via its public interfaces, which Layer2 provides to surrounding system components. This phase is to verify that a system component fulfills requirements in terms of functionality and performance. It also verifies that the component has an adequate maturity to be promoted a higher level of testing.

Layer2 SCT tests are written with the Robot Test Framework. It has a simple plaintext keyword-driven testing approach and it is easily extended with Python libraries. Robot tests and keywords are meant to be simple and straight forward

readable English prose. When going deeper into a programming test logic, the implementation fluently switches to the Python language. The Robot Framework also offers a possibility to attach tags on tests and suites. These tags can be used then further on to group tests with different topics, e.g. “shock” or “regression” –suites or to group feature related tests. Tags can also be used to identify non-critical tests which are failing at the moment for a good reason. When the test material is large, it is also useful to be able to tag the tests that are identified to be unstable. In this way an unstable test is still included in the test runs and a possible stability problem is visible in the recorded test history, yet still maintaining the test suite itself stabile.[7]

The Layer2 SCT environment contains several small in-house developed Python libraries to pull out a programming logic from the robot keywords. These Python libraries offer simple interfaces on storing the eNB configuration during the test, and various other utilities, which require a real programming language.

An open source network protocol testing library Rammbock is used to communicate between the test PC and eNB. It has a simple syntax on defining the protocol and messages. From the tests the messages can be then sent and received without distracting the test itself with messaging details. Also, when expected messaging scenario is not happening, the Rammbock can raise a descriptive exception about what went wrong [8]

From a system component test suite point of view, the system under test is accessed via the UDP and TCP communication. There are no dependencies on what hardware the SUT is running on, as long as it has an IP address. Thus all the existing SCT tests should be runnable also on a simulated target.

3.4 Test On Real Target – set up

The current SCT set up with a real eNB device contains one eNB attached directly via a router to one dedicated test PC (Figure 5). This test PC is used to run selected tests with the Robot Framework responsible for downloading a new software to the eNB, running the tests and collecting test results and logs and possible crash dumps from the target.

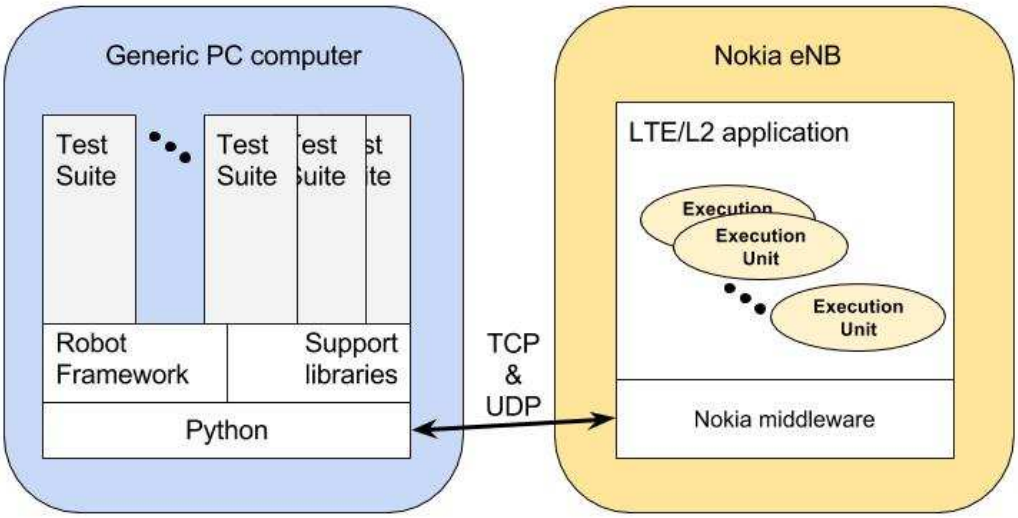

FIGURE 5. Layer2 SCT setup on real eNB hardware

These test lines are replicated to create pools of similar test line setups. Queuing a test line from the reservation server is the first step when starting the test. The test line reservation server takes care of balancing test line resources between different users (a continuous integration automation and developers) and stores information about test line usage statistics.

4 MOVING FROM REAL TO SIMULATED

4.1 Test On Host – set up

The Layer2 implementation is mainly pure control logic and it does not depend on many hardware services. From this perspective, it is a natural decision to compile and run the Layer2 application in the development host PC instead of simulating the entire target processor chip (Figure 6).

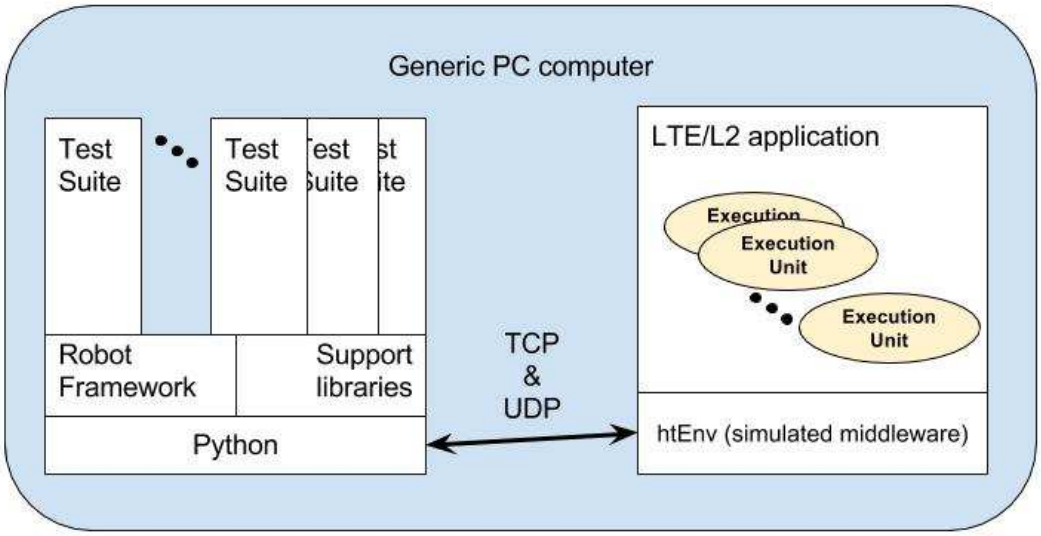

FIGURE 6. Layer2 application running on the host PC environment.

4.2 Compilation and linking with native compiler

The first task was to perform a compilation with a native instead of cross – compiler. This was actually simpler than expected. From various reasons the sources of a embedded software tend to attract pre-compiler directives and Layer2 is no exception. Most of the time in this task was spent on finding the correct combination of pre-compiler flags to include important pieces of code and exclude unwanted sources.

When linking on a simulated environment, all the function calls to an operating system and hardware support libraries will need to be fake or dummy imple-

mentations. For this purpose most of the OS service fakes were already available from a similar host environment of another system component.

4.3 Replicating the deployment

On a real ENB, the Layer2 application is running distributed on a multiple cores and multiple CPUs. Each of the EUs has their own start, stop and message reception callback functions. For different eNB products, there is a possibility to make several different target binaries of Layer2. One of the oldest and most stable “four DSP processor deployment “was selected to be the first one. In the end all the deployments should be replicated.

An existing middleware simulation for the host environment, which was adopted for this project, provided a solution to start these EUs as a threads of a Linux process. With a simple array definition, EUs address could be faked to be located virtually on a different CPU core.

Problematic here was the Layer2 application SCT setup, which contains parts of the same code compiled with different precompile flags. Thus in the end the test environment starts three processes sharing some pieces of shared code and these processes then contain threads replicating the real ENB execution units. Communication in between these processes is done via a pair of connected sockets.

4.4 Implementation tasks, the missing pieces

4.4.1 Python virtualenv

On a real hardware test line the test control PC must, have specific versions of Python libraries and tools. This is easy to fulfill as the test control PC is an isolated and dedicated device just for that single purpose. Instead, when running on a development host environment, the test control PC functionality has to be carried out on a server to which users cannot have admin rights. In the end different revisions of development history must be able to reproduce tests independently parallel on the same physical machine.

Versioned Python Virtualenv offers a solution for this problem. Virtualenv is a framework which makes it possible to create an isolated Python environment that contains exactly the requested revisions of Python packages. This environment can be then activated from a shared network folder mounted on a Linux system, and it will temporarily override installed system packages.

4.4.2 Runner script

On a real test line, a base station reboot in the beginning of a test session will guarantee that there are no hanging processes from the previous session. On a host environment it is likely that after a test run either the test runner or the tested application is not terminated properly. To keep track started processes and properly kill them at the exit, a new wrapper script was needed.

Afterwards this starter script was also utilized to introduce dynamically allocated port numbers between the tested application and the test framework because in the real environment everything, including TCP and UDP port numbers are dedicated just for one single test execution at a time. Now this is not the case when running several test runs parallel on a crowded development server. To solve this problem the whole test material needed to be modified to make it possible to switch the TCP and UDP communication to dynamically allocated ports.

4.4.3 Communication between tester and Layer2 application

One of the largest single tasks was to reverse engineer how the communication between the Layer2 application and test framework works, i.e. how to make it happen in between the test framework and a simulated target messaging queue. ENB asynchronous messages have a simple, always big endian, message header which in turn specifies the endianness of the message payload. The confusing part was that the tester and the Layer2 application are able to handle most of the messages despite the endianness. But for some of the messages it matters.

This problem caused a segmentation fault on the Python interpreter when receiving one specific message from SUT. This message contained a correct length in the message header but the length property of a dynamic array was

too big because of the wrong endianness. Because of that, Rammbock tried to decode more array elements than actually was received.

4.4.4 Timers on non-real time environment

When running some of the tests on the host environment, it became obvious that the development host PC is far from the dedicated real-time environment. Other processes running on the same PC will slow down the Layer2 application unpredictably, while test suites are still measuring all timeouts based on a real clock. This will cause instability on some test suites. A solution will be to detach Robot Framework timeouts from a real clock and patch a Python time service with a module that interacts a time synchronization with the Layer2 application. However, this was not doable in the scope of this thesis work.

4.4.5 Overlapping memory areas

Due to the architecture and deployment of Layer2 application in the embedded environment, there are several processes on several CPU cores accessing memory sections which are located either in the processor cached or in an external memory. When this delicate setup was moved to run on Linux processes, each eNB process now running on its own Linux thread, it revealed a problem with global variables.

On the target deployment those global structures are in the internal memory of separated CPU cores. Now when running processes of the Layer2 application as a threads of the same Linux process, all those global structures are shared between the threads.

Luckily, a C++11 –standard introduced a ‘thread_local’ storage specifier, which can be applied also on nontrivial data types. Most of the global data problem was solved by appending this specifier, though some of global variables had obviously no reason to be global and could be easily relocated inside of a C++ class instance.

Another alternative would have been to run each EU on its own Linux process. This alternative was discarded because it would have involved a lot of inter process communication making it harder to debug.

5 SUMMARY

The task was to enable running existing system component tests of the Layer2 application without the real base station hardware. The original assumption of the scope was to have a general suite setup procedure successfully completed. This setup procedure is a prerequisite for all the other tests and it is therefore the place from which forwards it makes sense to join more people and scale up the work effort on the host SCT ramp up. In reality, after a successful suite set-up, most of the tests were still failing on one same problem at a time. In the end there were not so many individual test specific problems. The scope of the work had to give in and it took a bit longer than expected.

Not all new implementations were made with unit tests. A tight schedule pressure and uncertainty on how big problems still lie ahead before reaching the goal, makes it really hard to be pedantic and consume time on tidying up and fiddling around with little details.

While working, some of the changes were done directly to the product trunk, but wider and larger changes had to be done in isolation at each own branch because it simply was not possible to predict what could broke with one commit. Merging between the trunk and this feature branch took a huge amount of time.

In the future constant maintenance will be required to adapt all changes of real interfaces to simulated counterparts. If the solution becomes valuable, then the maintenance shall not be a burden.

The work was considered to be done when most of the tests in the SCT shock suite were passing. Because of instability caused by the “simulated vs real -time timer” problem, 11 tests are still excluded. Some of the teams are already using this new way of running system component tests. Even if it is not yet perfect, it seems potential too for the daily workflow.

REFERENCES

1. Grenning, James W. 2011. Test Driven Development for Embedded C, Pragmatic Bookshelf

2. Rainsberger, Joe B. 2013. Video: Integrated Tests Are A Scam. Date of retrieval 15.5.2016 https://vimeo.com/80533536

3. Adzic, Gojko 2011. Specification by Example, 1 edition, Manning Publications

4. Engblom, Jakob – Guillaume Girard – Werner Bengt 2006. Article: Testing Embedded Software using Simulated Hardware. Date of retrieval 9.4.2016 https://www.researchgate.net/publication/250427239_Testing_Embedded_Software_using_Simulated_Hardware

5. 3GPP TR 21.905 version 13.0.0. Digital cellular telecommunication systems (Phase 2+)(GSM) Universal Mobile Telecommunications System (UMTS) LTE; Vocabulary for 3GPP Specifications

6. 3GPP TS 36.300 version 11.3.0. LTE Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN)

7. Robot Framework. Date of retrieval 23.5.2016 http://robotframework.org/

8. Rammbock protocol testing library. Date of retrieval 23.5.2016 http://robotframework.org/Rammbock/0.4.0/Rammbock.html