Master's Thesis

Requirements for an Interactive Logging Framework

Joakim Persson *D00*

Department of Computer Science
Lund Institute of Technology
Lund University, 2006

ISSN 1650-2884
LU-CS-EX: 2006-2

Master Thesis: Requirements for an Interactive Logging Framework

Keywords

Embedded systems logging, test requirements, interactive log tools, log data representation

Abstract

This thesis deals with increasing requirements for logging of embedded devices, especially mobile platforms. The main problem is how to handle increasing data flows from many independent software modules within a mobile phone, without sacrificing efficiency for both the embedded device and for the testers and developers. The challenge is the migration from the straightforward logging – with printed text strings – to a binary log format where the log information can be used to trigger logging events and to represent data in a more readable way, preferably with a standardized tool. Focus has been on log information that is related to signaling for WCDMA (“3G”) radio signaling, which can be hard to represent and can also potentially generate a lot of log data information in a short period of time.

The results of the experiments show both promising results, but also a need for reengineering the log tool framework on the PC side. There are problems with both maximum data flow and latency, which means that it will be difficult to use the existing interfaces for high-speed logging. However, on the bright side it is likely that the interfaces on the User Equipment side are sufficiently advanced to allow logging with performance enough to satisfy the test requirements, and the thesis finishes with the actions needed to achieve this.

Table of contents

1 Foreword

A few words about the audience of this thesis – the thesis is mostly suited to developers, testers and log tool designers within the EMP organization, as well as anyone with an interest in embedded software and logging solutions. Further, an interest in telecommunication as well as PC client/server programming is beneficial. Finally, the use of experiments, spikes and throwaway solutions can perhaps provide enlightenment for students of test-driven development, extreme programming, and other related topics.

Many of the details in this thesis, regarding proprietary tools and protocols, have to be omitted in this report due to confidentiality reasons. This is especially true for the “results” chapter, where quantitative measurements and such details related to the EMP and used third-party products are omitted. If these details are desired, please contact the NS Stack Test department at EMP. Still, the main tasks and conclusions are public and are included in this report.

2 Introduction

Large-scale software engineering projects always need some kind of assistance for test and debug of the developed software. This means that the role of tools and the toolsmith (see for instance discussions in [1]) is often underestimated when it comes to overall success for the project as a whole. The toolsmith is needed to create efficient ways of working for both developers and testers, so that the information that is needed to develop, debug and verify functionality is easy and convenient to access. This is the core reasoning behind the topic of this thesis, namely investigating ways to improve log tool functionality for embedded devices, or, to propose a new way to create an interactive logging framework.

The thesis has been carried out at Ericsson Mobile Platforms (EMP) [2]. The products encountered during the thesis work has been embedded devices, or “mobile platforms” (in practice “the most technical half” of a mobile phone or similar user equipment (UE)), which have special considerations when it comes to processing power, memory capacity, bandwidth and power requirements. This means that there is only a limited amount of memory and CPU that can be allocated to log and debug information, and preferably the level of logging can be varied during test and development of new functionality. A very important consideration is that the embedded device must be testable. Many embedded devices are built from a hardware-centered view, but it is also important to consider higher-level software, what hardware interfaces are available for logging, how can the software within the embedded device be loaded and debugged, how can memory and other status information be retrieved when something goes wrong, and so on.

At the company, a part of the product architecture is dedicated to debug and logging. The most basic part consists of a “PrintServer”, a module which is dedicated to handling log information sent as strings from different modules in the platform. The more advanced logging part is a protocol of its own, called TVP (Test and Verification Protocol), which is a binary format for sending log data. This is much more efficient and means that more logging can be performed for the same amount of CPU, memory and bandwidth. Basically, the setup can be viewed as follows, with one UE, USB for communication where log and user data is sent back and forth, and one Host PC for receiving the information.

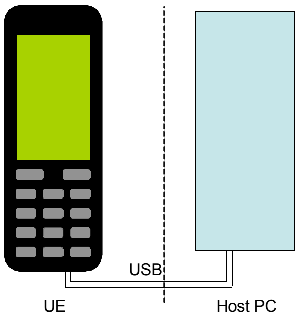

Figure 1: UE and log environment

The PC Applications used today are both produced internally at EMP for specific purposes, but a major commercial log tool is also increasingly being used. The main log tool handles both TVP and PrintServer information. A drawback of the main log tool is that it is not possible to rebuild the tool when new functionality or new TVP log points are added to the software (as it is third-party software), which severely limits its usefulness during the development phase. Further, the internally produced log tools all mostly rely on Print Server logging, which is becoming more and more inefficient as the volume and rate of data which needs to be logged increases. The solution would be a tool which could both use powerful TVP logging, but also a tool which can be flexible enough to add and remove temporary log points for debug information rapidly (for instance, between different platform software builds).

All this leads to the main practical focus of the thesis – how can an interactive logging framework be built, based on the architecture already in place? The creators of the main log tool have implemented a COM interface, where it should be possible to extract arbitrary log point information and handle them in a separate application through this interface. This means adding yet another protocol and application to the architecture, but has the benefit of testers and developers being able to migrate to the main log tool while still being able to activate and view debug log point information in a new application – combining the strengths of both logging the old-fashioned and flexible way with the more efficient binary type of logging.

This thesis explores the options available for realizing the “new log tool” part of this architecture, given the requirements and needs of the network signaling test organization (NS Stack Test) at EMP. There are many organizations with different logging needs (for example, Multimedia or Operating Systems) within the organization, and this thesis is restricted to the network signaling parts, which is large enough (over 100 developers and 70 testers) to motivate careful log tool investigation while not covering a too broad scope – some general background is given with regards to WCDMA radio signaling [3], which is very complex and needs to generate a lot of information in order to debug and understand what is going on.

A brief study has been made generally about how to integrate logging and debugging with other major development suites, such as Eclipse [4]. The investigation and tests of the new log interface has been carried out mostly through the use of experimental programming in a light-weight, high-level programming language, to be able to test as many alternative ways as possible to collect, represent and save log information. The experimental programming has centered around data transfer and parsing, and has explored options provided by Python [5], Java and C++.

The thesis begins with a discussion regarding the problem context – what is the environment like where the thesis results can be used, and what is the situation today (when the thesis work started)? The chapter “problem description and analysis” contains discussions and digs deeper into the reasons why a change is wanted. The chapter “results” contains the findings and prototype implementation, using the problem analysis as input. After this, the results are discussed and solutions proposed. The final chapter contains general conclusions, written once the entire thesis work was completed. The interested reader will find more literature and explained terminology in the appendix.

3 Methodology

A thesis such as this, dependant on a lot of experimental programming and exploring the problem space through prototyping, has a somewhat different kind of methodology connected to it. It has been my intention to try and explore many different solutions, rather than focusing on one single implementation. Therefore, the thesis work has been driven by changing requirements, observing the strengths and weaknesses of existing tools, and trying to apply the lessons learnt during observation to the environment available.

A solid methodology is needed to prove that the work process is robust, provides results that can be verified, and to make sure the conclusions and discussions about the results can be generalized and validated. That is, the scientific goal of this kind of work is to ensure, as much as possible, that engineers further studying the subject can learn from the work in this thesis and depend on the conclusions, and hopefully avoid the pitfalls and limitations encountered.

The methodology centers around light-weight, experimentally driven development. This way of working was initially chosen, since the problem description was intentionally vague. Exploring the current log environment and trying to adapt a new log interface to current ways of working meant that the process consisted of learning the details of the protocols, trying to activate the log points associated with the protocol, and then trying to parse and present the log points to see if it is feasible to include this kind of representation in a future log product. Each of these iterations were to be rapidly executed, once the basic interface was working.

A drawback of the methodology was that even though progress was good initially, it is still easy to become reluctant regarding abandoning trails. At first, the thesis was thought to be heavy on quantitative issues, but lack of time has meant that the results and prototype work has focused on more qualitative issues, answering questions such as “is this the best way of doing things?” rather than “exactly what is the throughput, latency and memory benefits?”.

3.1 Experimentally driven development

The main topic of interest during test development was experimentally driven development, which basically means that the software development should be performed iteratively and with the aid of testing (preferably unit tests or module tests) while development proceeds. Each development cycle is short, to be able to properly evaluate the progress or lack of progress, so that there is always a possibility to step back.

This is to be seen in contrast to more heavy-weight development methodologies, such as RUP or other models with larger iterations. This is the dominant process within the company studied in this thesis. The reasons for not choosing to conform with a top-down requirements analysis, design and implementation is that the end result was not ever expected to be a complete application – but rather a collection of prototypes and ideas that can be used within the company to improve the tools. Therefore it was decided that smaller prototypes developed rapidly was the main method to solve the thesis problems. Performing the project completely separated from the projects within EMP also meant that a lot could be learned by observing test and development while the thesis was being written, without participating.

3.2 Extreme programming

The concept of Extreme Programming (XP) [6] fits well into a lightweight methodology. XP has become famous for its unorthodox way of looking at software engineering – more like a sequence of very short iterations, where each iteration is executed through test-first development (that is, writing test cases before actually implementing the code) and simple design, among other things. This is in contrast to more formal, waterfall-like models, where there are always separated phases for requirements collection, design, code, and test. The decision to use a kind-of XP methodology was taken since the test-first, simple-design way of working seemed like a good fit together with the task.

The way XP is used in this thesis is mainly as support for the development process – by putting up short-term goals and attempting a lot of experimental spike activity to learn more about the system. Since the whole system is complex and dependant on many external factors, it is important not to lock development to one single train of thought. Instead, rapid feedback (from testers, developers, and tool creators) should be allowed to change the requirements and design once work progresses.

3.3 Language choice and tool development

In practice, there are several ways of developing software within a large organization. Within EMP, software is either developed targeted towards an embedded device (for instance an ARM embedded processor or a specialized ASIC), or targeted towards a PC environment. In most cases, C or C++ is the language of choice for all kinds of development. Some PC tools have however started to be developed in more light-weight languages, such as Python, but most tools are written in C++ or Java. Apart from programming languages and IDE:s (Eclipse is the preferred choice), version control is essential for all development as there is a huge amount of versions, branches and variants for the software – this also impacts test cases, which need a similar version control system. Version control is implemented using ClearCase and an additional configuration layer, called CME.

Traditionally, NS Stack test tools also obey strict version control. All scripts and tools need to be controlled this way, in order to handle different variants (for instance, for different hardware architectures or network vendor configurations) with as little maintenance as possible. Any new tool should preferably fit well into this configuration system, which means that it is better to create tools that are small and open, rather than large and monolithic.

This has impacted the methodology of this thesis. In order to achieve a maximum fit for a prototype within the test tools organization, it would have to be written in C++. However, the effort involved in doing this would far exceed the 20 weeks that were at the disposal for this thesis. Also, not all possible options for exploring the ideas and new interfaces would be covered in this time if everything was to be done in a high-level, “waterfall” mode.

Instead, a different development choice was chosen. Python was chosen as the main development language, as this is a light-weight, high-level language that still contains all the relevant pieces needed for completing the project.

As an option, Java programming was considered, to try and perform the same tasks there. And finally, should both Python and Java seem inadequate, C++ is to be used – most other tools are developed in C++, as well as the main log tool and a very brief example program using the Bridge interface.

4 Problem context

In order to understand the complete system that the logging problems involve, it is necessary to present some basic facts and information about different system components. Since this thesis was written at EMP, the focus has naturally been tools for embedded devices. Exactly how these devices operate is of course confidential, but the basic ideas can be outlined to show why logging is not such a straightforward operation as it might seem.

The users that come into contact with the products from EMP can be roughly grouped into three categories – developers, testers and customers. Each group has different needs, and this needs to be reflected by the test tools.

4.1 Embedded devices

To start with, embedded devices are any kind of computer that resides within the device that it controls. Embedded devices place special considerations on the designers and testers of such devices. The main obstacles of embedded devices that are usually not taken into account for normal computers are as follows:

- CPU and memory architecture limitations: Embedded devices are manufactured in hundreds of thousands of units, and the cost constraints placed for mass-market products means that all CPU and memory solutions must be as cheap as possible. This means that the designers must be careful not to waste precious CPU and memory resources during development and deployment of the product. The impact this has on testing is that test functionality takes CPU and memory resources that could have been better spent doing real work.

- Power limitations: Embedded devices are typically battery-powered. CPU- and memory/disk-intensive tasks consume a lot of current, and this must be minimized as much as possible. If a test or log process is consuming a lot of CPU, it will naturally also drain power from the battery, which in turn will shorten battery life – a very undesirable side effect for data which is only needed by a limited amount of people.

- Interface bandwidth limitations: Embedded devices cannot always be logged by software running on the same device. This means that some kind of interface between the embedded device and a stationary computer is needed to perform certain kinds of logging.

- Hard real-time constraints: Embedded devices often require an OS with deterministic scheduling, to make sure that important tasks are performed within a certain time. This is especially important for communication devices.

Taken together, the task of programming and testing embedded devices is usually more complex than ordinary PC applications. For instance, ideally you would want to be able to step through the instructions of a device with a debugger to catch exactly where unexpected situations or memory leaks occur. For embedded devices, this can be quite hard for real-time situations – the debugging process itself can throw off the timing of the processes.

Apart from straightforward logging from different processes in the embedded device, it is also important to have access to the memory contents of the device. The ability to perform and capture memory dumps is important to be able to reproduce and solve software errors.

4.2 Mobile platforms

A mobile platform, in the EMP perspective, is basically the “technical half” of a mobile phone. For instance, the mobile platform is a blueprint, combined with hardware and software, that enables customers to build their own phones. A mobile platform in itself is not of much use, so EMP also produces reference designs to show how the platform can be used, including both hardware and application software. This reference design together with the platform software is what has been studied during the thesis.

A mobile platform can be seen as hardware chips (for instance a baseband processor together with some specialized ASICs), together with software that can basically be modeled as stacks – for instance, a network signaling stack, a data communications (such as TCP/IP) stack, and so on. This is a fairly natural way of viewing communications software, and for this thesis, it will provide most of the conceptual visualization needed.

The actual terminals are known as User Equipment (UE) in the standards organization Third Generation Partnership Project (3GPP). A UE consists of many interfaces, some of which are useful for testing. For instance, a UE usually has a keypad and a display for navigation and user input, but more importantly a UE also has a USB, RS232, Bluetooth and/or Ethernet interface in order to move data from the UE to another device. For network signaling test purposes, it is also important that the UE has an external antenna interface (so that the antenna can be replaced by an RF cable connected directly to a base station or a base station simulator).

In a test environment, the testers would like to have as much control over the UE and the test equipment as possible. Since a controlled environment is important for validity and reproducibility of test cases, the UE and the associated tools and interfaces must work in a deterministic way. The antenna interface is easy to control, as it is a one-way interface, but the USB interface can be a bit harder, since the bandwidth is limited and the physical USB interface carries log data, UE commands and user data in both directions. A low-latency interface is desired, since many of the tests are time-critical and it is easier to debug if log data and UE commands can be synchronized as much as possible.

Often, it is enough to process the log information after it has been collected, but it is also valuable to be able to see what is going on in real-time or near real-time. Tests can be performed in the labs of EMP, but it is also important to execute tests at other network vendor labs or at type approval facilities, and the ability to trace the error while the test is being executed is valuable. Some tests are being performed on-the-move (in a car, high-speed train or other mobility environment), and sometimes it is important to pinpoint the location of the problem. So, there are different "environmental" concerns with regards to the nature of mobile platforms that involve logging. The temporal aspect (in which order and how often/for what duration) do things occur is most important, but state transitions and periodic measurements are similarly important, as are the mentioned location issues – where the test is executed can have tremendous importance.

4.3 WCDMA

Wideband Code Division Multiple Access (WCDMA) is the dominating "3G" standard for wireless communication. In this context, WCDMA and the protocols associated with it are the key research area at the department where this thesis was written. It is therefore natural to describe some of the characteristics of WCDMA, to show some of the complexities involved. WCDMA can be compared with older mobile standards such as GSM/GPRS and EDGE, which are TDMA-based standards.

WCDMA, on the other hand, is a CDMA radio access technology – this very briefly means that UE:s can transmit simultaneously over the air, spread and scramble the signal over a wide spectrum (5 MHz), and then the base station can despread and descramble the signal by using a special agreed-upon code. The details of the physical layer WCDMA procedures can be seen in [3]. Some special points of interest to logging might be that the physical layer performs very rapid power control, in order to counter fading and other radio phenomena. A characteristic of CDMA – where different UE:s can transmit simultaneously – is that the power level at the base station must be roughly the same for all incoming signals, and this means that the base station must control the UE:s output power frequently – up to 1500 times per second in the WCDMA case. This is also the smallest "unit" on the physical layer, known as a WCDMA slot (taking up 666.6 µs). 15 slots make up one WCDMA frame (10 ms).

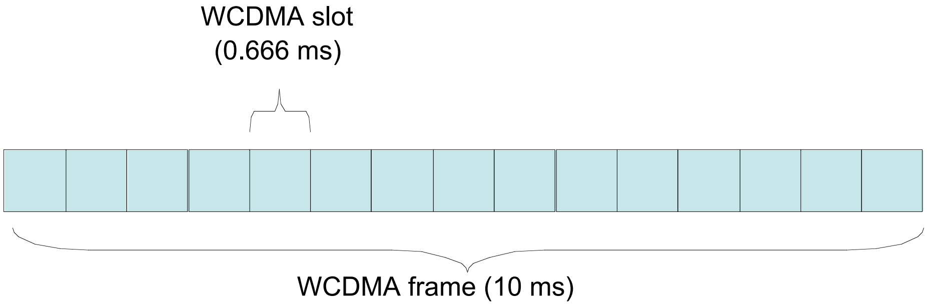

Figure 2: WCDMA frame and slot structure

These figures can be seen as an indication of what kind of resolution the different logging points could possibly have, something which is important to the analysis of the logging situation. Also, WCDMA is suitable for both voice traffic, data traffic, and a mixture of voice and data traffic. The different possible configurations (of different radio bearers, etc) must also in some way be represented. The actual user data (speech, video, packet data) could also be interesting to process – for the UE:s of today, this means that anywhere from 0 to 384 kbit/s of user data could be processed, and possible uplink and downlink speeds are of course projected to rise in the future. Note that because the same data is being transferred in different layers, it might not be sufficient to capture data in one layer alone (such as IP), but data from different layers might need to be collected simultaneously (both IP packets, the segmentation in RLC, and the physical frames), which multiplies the bandwidth requirement.

The UE does not live alone in an isolated world – as always in the cellular communications world, there is a large and complex infrastructure supporting the UE. This infrastructure is known as the UMTS Terrestrial Radio Access Network (UTRAN) in the 3G world. A view of how this can look like:

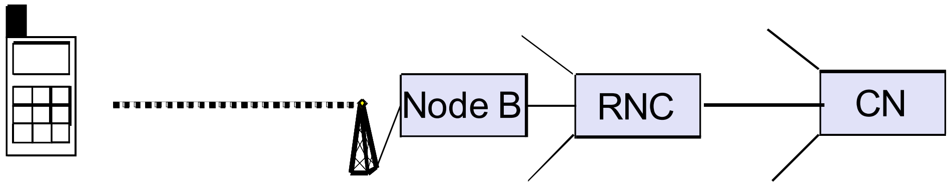

Figure 3: UTRAN main nodes

Here, the UE communicates directly only with the Node B (base station). The Node B is controlled by the Radio Network Controller (RNC), which in turn relays the user data to the core network (CN) – the user data can either be speech or SMS, and then belong to the circuit-switched part of the core network, or packet switched (e.g. WAP, e-mail, Internet access), which belongs to the packet switched part of the core network. For mobile platform manufacturers, the most interesting interface is the radio interface between the UE and the Node B / RNC (the “U_u” interface). The exact details of the UTRAN and the core network are not important for the rest of this thesis, but can be studied in [3] or [7]. The thing to recognize here is that there are different peer entities communication with the UE, and that logs taken on the network side of the UTRAN should be synchronized with the UE logs to debug things like timing and ordering of messages. Some messages are terminated in the RNC, while some messages are terminated back in some server in the CN.

As can be seen, there are many entities and interfaces involved in radio communications, and the desired state for the test organization is to have full control over every one of them. The following chapter will describe one of the protocols needed to control the communication in a WCDMA network.

4.4 Radio Resource Control and other radio protocols

An important subset of the WCDMA standard is the radio resource control (RRC) protocol [8]. This is the protocol which has been used for testing the conceptual ideas in this thesis. In order to understand the complexity and why it is worth the effort to construct better log tools, some explanations regarding what possible interesting things can be analyzed and debugged is needed.

For all telecommunications design, a good conceptual grasp of the Open Systems Interconnection Reference Model (OSI) is needed. This is the classical "seven layers" structured way of describing communication protocols functions. OSI divides the protocol functionality into layers that (in theory) only communicate with the layers directly above. A brief description:

Layer number	OSI Name	Functionality	WCDMA radio equivalent
7	Application layer	This is where the applications enter the protocol stack.	N/A, as this is represented by applications within the mobile platform.
6	Presentation layer	Translates application data (such as user input) into machine-readable format.	N/A, as this is represented by applications within the mobile platform.
5	Session layer	Allows applications on different computers to establish and use a session.	Typically handled by, for instance, TCP in the case of packet switched communication.
4	Transport layer	Handles error recognition, recovery, and segmentation. Repackages long messages into shorter fragments.	Typically handled by, for instance, TCP in the case of packet switched communication.
3	Network layer	Addresses messages and translates logical addresses into physical addresses, handles routing and resource management.	IP is usually used. For the radio-specific network routing, RRC is used.
2	Data link layer	Handles the transmission of error-free, sequential frames.	The Radio Link Control (RLC) and Medium Access Control (MAC) are

			located on this layer
1	Physical layer	Transmits the raw data over the physical medium and regulates the transmission of data, for instance between the network card and the physical cable.	Proprietary physical layer (PHY) protocol is used.

The OSI model does not perfectly capture the specifics of radio communication, but a rule of thumb for network signaling is that all the protocols involved are in some way placed on layers “one to three” in the model. During practical discussions between testers, developers and customers this is usually divided into “L1” issues and “L2/L3” issues, to divide problems between physical layer problems, data link layer problems or logical (network) layer problems.

Understanding the OSI model makes it easier to understand points of interest during protocol negotiations, and such points are important in order to develop the probe and “log point” concept used as a metaphor during the tool development. From an end user (“higher layer”) point of view, for instance, a mobile phone is not very complex – the log points could be represented as the perceived state of the mobile phone. When the user starts to use the phone, it is in an “idle” state, not performing anything useful for the user (except being able to receive incoming calls and messages). Once the user has dialed the number he or she wishes to call, the user presses the “dial” button and perceives the phone as being in a “dialing” state. Finally, once the person on the other side answers, the phone is in a “talking” state. This entire flow can be visualized in, for instance, a state diagram[1].

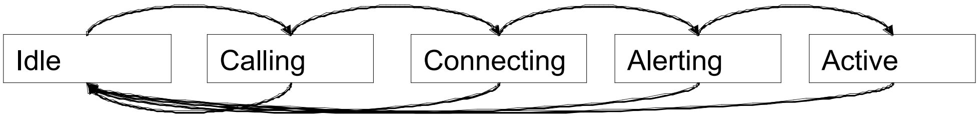

Figure 4: State diagram for a voice call (“call control”)

Apart from state transitions, typical functions for radio communication protocols also involve measurement collection and measurement reporting, as well as sending and receiving other messages for setting up and tearing down radio connections. This is what happens on the RRC layer, which can generate messages and measurements at a considerable pace[2]. Also, before the control data is actually sent over the air, it must be fragmented, sequentially controlled and integrity checked by the lower layers (RLC/MAC) and properly sent over the air (PHY).

[1] The “call control” state diagram here also contains additional states important to the network – a UE in a “calling” state has, for instance, not yet received confirmation from the network if the other UE exists at all or the number dialed is valid, while in the state “Alerting”, both UE:s play dial tone sounds.

[2] The RRC protocol is specified by the 3GPP organization, and the specification (25.331) contains over a thousand pages in the latest version.

5 Problem description and analysis

The importance of logging and representing log data has been described in the previous chapter. Some of the key points have regarded the full complexity of the system – network signaling requires a lot of information for debug purposes. Another key point is that many things in the air and in the software occur at a very rapid pace – this means that it is important to timestamp and trace different events and measurements to be able to recreate the situation. As the target platform is an embedded device (a UE), it is also important that it is testable and that the logging itself does not introduce a different behavior in the UE (such as degraded performance or different timing since the logging processes consume CPU and memory).

This is the input used for defining the goals of the thesis, of wanting to construct better tools than those used today. In this chapter, issues that were important to the problem are discussed and analyzed.

5.1 Logging and embedded devices

Logging of embedded devices can be implemented in several ways – the log data can either be stored on the embedded device, to be extracted in a future moment, or immediately be transferred to a host device (such as a PC). This thesis will only look at logs transferred to the PC, and how they can be processed. The reason behind this is that there is currently no implementation that writes log information directly to the UE, although that would certainly solve performance issues – for instance, by writing log data to a memory card and extracting this later on for post-processing.

Currently, developers implement all the logging, and the testers observe and analyze the logs that are generated during test execution. The main part of the log framework implemented by the network signaling modules are implemented using printf() functions in the source code. These functions transfer the strings to the PrintServer module, where the prints are later sent with the DebugMux over USB (or, in some cases, sent over RS232).

The ownership of the actual log prints is completely in the developers hands. This is something which is inherited from the traditional project way of working – designers develop and implement new functionality, which are then normally transferred to the testers as-is in the form of complete builds. There is no clear distinction between log prints that are harmless, part of normal procedures, warnings or errors. This has lead to a situation where different groups focus on different kinds of procedures and error conditions, and where there exists some tools for post-processing logs to filter out warnings and errors. Since the number of log prints is vast, it can be difficult to spot "side-effect" errors through manual inspection, unless the test case exposes the error. For instance, a voice call might be perceived as successful by the tester, even though later inspection of the logs show that signals are unnecessarily sent many times or some procedure is not executed as optimally as possible. All in all, leaving testers outside the task of specifying and discussing what kinds of prints and logging should be implemented can be dangerous for the future, as much of the information regarding why a certain print string is implemented could be lost if nobody within the test departments knows why it is present.

The amount of prints also means that there is no way that all prints can be enabled at the same time. This has lead to a categorization of prints as "A-prints", "B-prints" and "C-prints", where "A-prints" are always printed, while B- and C-prints can be turned on through an interactive debug facility or by rebuilding the platform. The question of which prints to enable or disable is mostly determined through experience and of which modules are being used in the test case, or through analysis of incidents found during test.

This all boils down to the question of whether the current log concept works today – and the consensus at EMP seems to be that although "printf()" logging has some advantages (easy to implement, powerful for experienced testers), it also has many disadvantages (performance and flash code size). This means that a reevaluation of how to log is needed, and that should also include a reevaluation of the roles of testers and developers (and also possibly customers).

There exists an option to PrintServer logging in the UE – namely the Test and Verification Protocol (TVP). This protocol is used by some departments to transfer log data from the UE to the PC in a much more efficient way – instead of sending raw strings to the PC, the log data is sent as packets using a standardized protocol (with header and payload), and the processing burden is thereby placed on the PC side. This protocol and the tools associated with it are used extensively by some departments, but not as much by the NS stack and NS stack test departments. Part of the reasoning behind this is the sheer volume of existing prints and the effort it would take to refactor the code to use TVP log points instead of printf() functions. The main objective of this thesis is to investigate if existing log tools can be expanded to combine the advantages of the two log formats.

Both Print Server data and TVP data is transferred to a data link module, known as the DebugMux server. The DebugMux server packages the information, and sends it over USB to the host PC application. The host PC application contains a DebugMux client, which unpacks the data, and then the data is ready to be used. A simplified view of the architecture can be seen as follows:

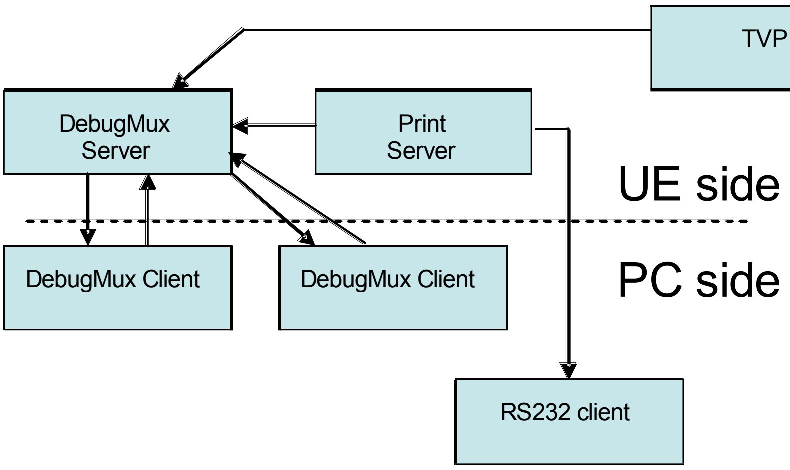

Figure 5: A basic view of the modules required for logging EMP products

The information that is sent through the Print Server is in the form of simple strings, generated by calls to the standard printf() function in the C programming language. This means that printf() strings can also be sent raw through the RS232 interface (serial cable), which is a simple way to log what’s happening. On the other hand, TVP packs the log point information into binary data structures and transports them through a data link layer (DebugMux), guaranteeing lossless delivery. The picture below shows the basic structure (note that Print Server strings are usually sent through the DebugMux layer these days, with the RS232 option rarely used as a residual option):

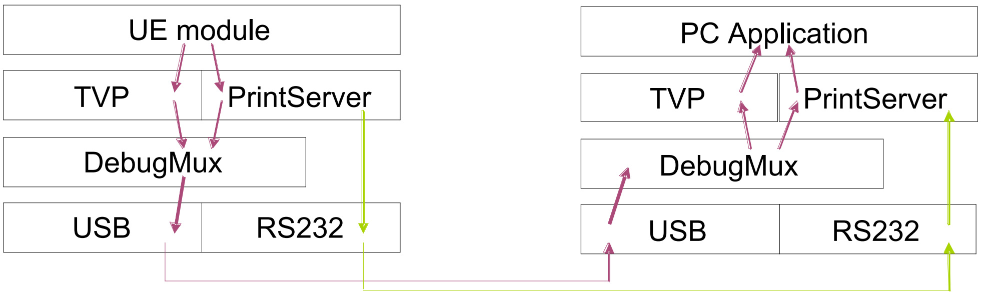

Figure 6: Overview of protocols used for logging, dark arrows indicate the main log paths, light arrows indicate redundant log paths

5.2 Role of testers and developers

In a large organization such as EMP, the roles of testers and developers are separated. This means that the system itself is complex enough to warrant a separate organization for test development, test execution and test reporting. Tests are not separated at all levels, as the developers tend to design their own tests for unit and module testing, but function and system tests are owned by the test departments. This means that the role of the testers are often to verify system requirements, perform load and stress tests, perform interoperability tests against network vendors and operators, and perform internal type approval tests. It is important that the logging framework is mutually beneficial for both the developers and the testers.

When introducing TVP log points to replace the PrintServer log points, it would therefore be important to agree upon exactly what kind of contents must be included in each log point, and how this log point is optimally represented. This means that some sort of cooperation, for instance during requirements specification, should be made regarding how test and verification of the functionality should work. Testers should be involved in the review of requirements specifications and implementation proposals of functionality which requires new log points, to make sure that the representation of the log data can be developed in parallel with the functionality.

The reasoning behind this is that there could otherwise be a risk that TVP log points are implemented without including the data that is important to log – TVP might be implemented without enough regard to the ability to verify the functionality. If testers are involved in the specification and design of log points (and also involved in the design of the new log tool), there is a greater chance of doing the right thing. This is more of a project management issue than anything else – one might also argue that the scarcity of resources means that the testers should do what they do best, namely using the existing functionality and system, finding the bugs, and aiding with the debugging, and thus leave all the implementation details to the developers.

5.3 Tool support for logging

The mentioned architecture within the UE and the definition of the logs would not be worth much if there was no good tool support on the PC side for logging the platform. As it turns out, the protocols implemented on the UE have peer entities on the PC – the functionality of the PrintServer is easily implemented on the PC with the help of a good windowing class, the DebugMux server on the UE has a corresponding DebugMux client on the PC, and the physical transmission is made through the use of USB.

There are two tools used for logging – one EMP-made product for logging PrintServer logs, and one third-party product (“the main log tool”) that logs both PrintServer and TVP (a selected subset of TVP points). The main log tool includes many desired features, making logging easier to understand as well as dividing streams of information into different windows. The main log tool is also delivered to customers, to aid them in their own debugging.

5.4 WCDMA protocol logging

The information in chapters 4.3 and 4.4 has served as important input regarding how to log WCDMA protocols. In order to analyze this situation, some clarification is in order regarding how the system behaves. The easiest way to visualize complex communication protocols is to draw layer diagrams, using the OSI model. For the relevant WCDMA protocols, the protocol stack (encompassing layers 1-3) can be visualized as follows:

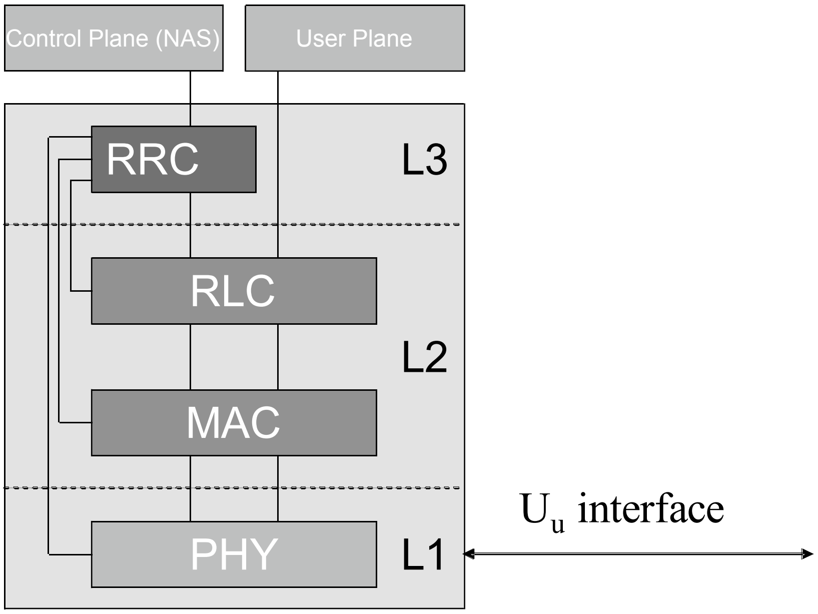

Figure 7: The network signaling protocol stack

Thus far, only the UE side of things has been discussed. Each protocol layer also has a peer entity on the network side. For instance, the RRC protocol is terminated in the RNC (see figure 3). Being able to synchronize logs on both the RNC and the UE is extremely important, as many issues can be related to timing, retransmissions or messages being sent and received in the wrong order. Also, things like ciphering (encryption) can complicate things further. If RRC is taken as an example, for these reasons, it should be possible to log both the actual control message (RRC), over which radio bearer and logical channel the message is sent (RRC -> RLC), how the message is fragmented (RLC/MAC) and exactly how the message is sent over the air (PHY). Each of these layers generate data at a different pace, and this means that log points can and should be inserted at the different points of interest in the stack.

For demonstration purposes, this thesis has studied measurement reporting. In WCDMA, measurements are performed by PHY, and the results are reported to RRC. If a voice call is active, then PHY performs measurements every 70 ms, and if certain thresholds and conditions are fulfilled, measurement reports are sent to the network where addition of a new cell is desired. Without going into more details, this scenario is good for observing operations at all interesting protocol layers, including internal operations within each layer (in this case – PHY measurements).

Recall the resolution and amount of log points desired, as described in chapter 4.3 and 4.4. It seems like it should be possible to get data on a frame level (10 ms resolution), but logging data on the slot level (0.666 ms resolution, or 1500 slots per second) seems a lot more difficult – a worry is that such a high level of logging strains the CPU on both the UE and the PC too much. The implemented log points studied also do not usually include slot-level resolution.

5.5 COM programming and communication

COM communication is one of the main means of inter-process communication in Windows environments. A lot of information regarding COM can be obtained from the specifications in [9]. COM is basically about exposing different kinds of interfaces between different programs – the actual object model of each COM object is hidden from view.

COM is used, for instance, between many of the common Windows applications, such as Office, Matlab, etc. COM interfaces come in different flavors, where the simplest one is known as the “Dispatch” COM interface. For users of a dispatch interface, there is no need to know exactly how the invoked program is made or what exactly is executed, which makes it easy to call standard functions without knowing how they are implemented.

For the log environments at EMP, COM is used for communication between the main log tool and the new Bridge interface. Initial analysis of the interface showed that the task of implementing calls to the interface should be easy – but after a lot of testing, each interface call worked except for one non-standard COM interface call. The Bridge COM interface is not implemented as a standard Dispatch interface, but rather as a kind of “Dual” interface which has less extensive support in languages other than C++. Eventually, this led to the solution that all COM communication was performed in C++, while all parsing and representation was performed in Python through the use of shared files. This should later be replaced with other COM interfaces or named pipes for inter-process data transfer.

Recall that both the main log tool and the specifications of the Bridge interface are already created – and a basic implementation of the “Bridge Client” was used initially before creating other prototypes. Also note that the “main log tool” implements the DebugMux client, as can be seen in figure 6.

5.6 Why is a new log tool needed?

The major issue regarding the tool situation is that there is no utopia. The PrintServer logging is easy to implement and has decent tool support, but is somewhat unreliable and wastes resources in both the UE and the PC. TVP logging is already partly implemented, but the main log tool used for this is not owned by the EMP organization, but instead developed by a third party. This third party does not deliver updates to the main log tool very often, and this means that the procedure of adding new log points cannot be performed on a day-to-day basis, as with PrintServer logging. In order to combat this situation, EMP has requested and received a new interface to the main log tool, where arbitrary log points can be activated and deactivated, and thus controlled by a new log tool. The investigation and prototyping of this new log tool and the “Bridge” interface has been the practical focus of this thesis.

Summary of characteristics of the main log methods:

PrintServer logging	TVP logging
Flexible, easy to implement new log points in the UE (just add a printf() line)	Implementing new log points in the UE requires more effort in a TVP module, but the end result in the regular software modules is just one extra line
No effort required to print additional prints on the PC	Complete rebuild is needed of the main log tool if a log point is added
The main processing and memory burden lies on the UE	The main processing and memory burden lies on the PC
A high level of activity can cause print overflow	Log points are sent as packets over a reliable connection-based protocol
Most prints have to be added through rebuilds of the platform, some can be added or removed during runtime	TVP points can be activated, deactivated, stimulated (to execute “target test code” in the UE), and several other tasks
Actual prints in the source code can take a considerable amount of valuable flash space in the target device	A separate module in the UE for handling TVP information is needed, but less data needs to be stored as there simply are no print strings

As can be seen, TVP looks like the more attractive protocol, but the main disadvantages means that TVP points cannot be added or removed on a day-to-day basis, something which is absolutely essential for new development. This is what the new Bridge interface is supposed to relieve – effectively removing this drawback, and also enabling the NS Stack Test department to implement new tools to represent log points not only through text strings but also through other means.

Debugging software often means changing the behavior of the software. A test case which fails can often be troubleshot by adding prints, but to see if changes are effective, it must be possible to change the functionality as well. This is usually of course done through rebuilding the software, but during software development it would be very beneficial to do it without rebuilding. For instance, a typical situation could be tests performed at a network vendor lab during a limited period of time. Spending hours rebuilding the software is an unnecessary waste of time, if the same kind of functionality can be switched on and off during run-time. Of course, this is not practically possible for many features of the software, but things like parameters and alternative paths of execution should be possible to trigger and/or change without rebuilding. This is something which is partly possible today – the “interactive debug” facility is available for some modules, and the TVP points can be “stimulated” and thus execute target test code in the UE, but this is not yet possible to do without special software builds.

5.7 Other tools

Software testing does not merely mean executing tests and logging/analyzing the results. There are a range of other tools needed to fulfill the tasks of the software departments. A point of interest here is how these tools are affected by changing the log architecture, and if the tools can somehow be integrated. For example, EMP has recently switched to the Eclipse IDE. Eclipse is an open-source IDE which is basically just a shell for different software development tools, such as a C++ compiler, debugger, software version control tools, and so on. Other tools that could be included in Eclipse in the future include for instance logging tools and software loading tools. A brief analysis has been done regarding how this can be solved – the main log tool used for connecting to the UE must in this case be started, but the new log tool could be integrated into Eclipse. Eclipse uses java for all kinds of development, and it is rather easy to incorporate new plugins into the Eclipse environment.

5.8 List of problems and requirements

The problems described in this chapter can be summarized in the following list, which has served as input for the main work of the thesis:

- TVP logging from an embedded device means that performance, in terms of latency and bandwidth, is important for PrintServer to TVP-migration.

- Because TVP is a binary protocol, the binary information must be transformed into readable information by the PC application.

- The representation should take several different forms, to accommodate different types of logging. Printing strings is the only option available now.

- The representation should be a collaborative process, so it should not be too difficult to add new log points with the same format.

- COM inter-process communication is a constraint, since this is the chosen method for communicating with the main log tool.

- TVP and DebugMux are also constraints, although these protocols are under EMP control – still, for this thesis the current TVP and DebugMux implementations must be treated as constants.

- For debug purposes, the tester should not have to rebuild the entire platform, but rather change parameters and software behavior during run-time, in the standard build variant.

6 Results

In the previous chapters, the problem context and initial problem description and analysis was described. This chapter contains the results of interviews, experiments, and evaluation of tests using the newly developed prototypes. As has been written, there are several requirements that cannot be changed while analyzing and trying out solutions to the problems.

6.1 Requirements for log tools

By starting from the beginning, with collecting requirements and thoughts over what testers and developers actually need to perform their work, an exhaustive list of possible requirements was collected. This eventually led to a list of requirements, which is not included in this thesis but which served as input for the prototype work. For instance, it is important that the tool is easy to use and gives a clear representation of what is going on. The current PrintServer style of logging requires quite a lot of experience and training until it can be properly used – mostly because the myriad of log prints means that it is nearly impossible to see in real-time what is going on, and even post-processing the logs can take a lot of time. Using PrintServer logging, all prints are mixed (row by row), so different modules and stacks all carry on printing at the same time, which means that it can be tough to see the procedures clearly. The situation becomes severe when prints other than the A-prints need to be added, as this adds to the clutter of the main log window.

There are of course workarounds to this, but most involve post-processing the rather large logs to make them more readable, for instance by filtering out prints that do not belong to the network signaling stack. The sequential way of logging is also not a very natural perspective of what is actually going on in the code, and thus is not optimally suited for debugging. Although the temporal sequences are well suited for PrintServer logging, radio network signaling is very much about state machines and measurements, and it would be more natural to view this as one single variable or state being changed, rather than a lot of prints each time the state changes. For instance, if the tester wants to know what state a certain module is in a certain point of time, it is likely that the tester needs to scroll or otherwise parse through hundreds of log lines to find out the current state. The protocols and procedures can in turn depend on up to ten different state variables, something which is very time-consuming to look up. The same thing is true for measurements and triggered events. Procedures such as handover and cell reselection (when the UE decides to listen to a different base station) depend on signal strength and signal quality becoming greater than certain thresholds, and this would also be more natural to represent through for instance graphs or other means.

This input, taken together with the wish to be able to dynamically add and remove log data, meant that certain basic primitives were introduced to symbolize different kinds of log data. Before concluding that these were the correct primitives, some discussion and analysis was performed – the reasoning behind moving towards a binary form of logging instead of the straight-forward way of logging is that less information needs to be transferred from the UE to the host device. It was observed that it seemed like a lot of waste to let the UE handle the processing and formatting of the logging, when the PC (which is less constrained when it comes to CPU and memory performance) would be much more suitable for this. The PC should then in some way transform the binary data into a human-readable representation, and it should take the form of one or several outputs that can logically be connected to the internal operations of the UE. The primitives chosen were:

- **Raw output:** Exactly what is executed by the code – good for lower level debugging. This is always needed as a fallback solution, for instance to debug if the logging actually works.

- **Print output:** The basic means of logging today, print a string explaining what is going on – good for tracing sequences and function calls within the code. Print output is still very natural for the testers and developers today, and it seems to be impossible to abolish this form altogether, although a good performance enhancement can probably be seen if the actual printing is performed in the PC rather than in the UE. An idea is to for instance create a special 16 bit log point, only containing an index, which could then be connected to a generated file containing all static print outputs in the PC. Things like printing “entering function A()” should not be stored within the UE – rather, the UE should just send the 16 bit variable to the PC, which then maps the value to the correct string for output.

- **Graph output:** Draw a graph of a certain variable, represented by a log point – good for measurements, power levels, downlink and uplink throughput rates, etc. This is a very natural way of observing these things, and is also something which is used in the actual 3GPP specifications when describing certain procedures. This is also one of the main ways log points are used today within the main log tool, so it is very natural to want to duplicate this for log points not yet added in the main log tool. This is however also a more advanced implementation task, but the option should still be available in a final product.

- **State machine output:** Change a static image of a state machine to represent the changing states – good for state machines (the “talking states” shown above is much easier to understand if logged this way rather than with text strings that get lost among other strings). The reasoning behind this is that state machines are used in so many modules, and it is probably the thing that is most difficult to visualize since all the tester sees is a large amount of print strings being printed at a rapid pace.

- **Table output:** Change a static table – good when for instance debugging cell changes, handovers or other events that depend on several variables and their relationships. This could also be a cheaper way to implement state machine outputs – just display a table and change the variables accordingly when a log point is sent by the UE.

Other requirements include wanting to be able to rewind and replay the logging that occurred during the test execution – it was seen that a way to debug errors is usually to compare the execution with variants of the same software, older software, or older logs when the test case was working. It is also good to be able to take logs and insert them into other tools, such as Matlab or Excel, for post-processing. This is something which is possible but cumbersome with the current text-based logging, requiring much manual script effort (and also sensitive to format changes in the PrintServer output!).

For controlling the activation, deactivation and stimulation (injecting data into the log points to execute code in the platform), a GUI or console is needed, but for automation purposes this should also be able to execute through scripts. Another wish is to activate and deactivate log points based on events that occur – for instance, if something unexpected happens, the log application should have the possibility to sense this and add extra prints to capture more information. This implies a logical script language for intelligently sensing when something goes wrong, when a state variable makes a forbidden transition, and so on.

6.2 Architectural ideas

After having collected the requirements and gotten a view of what was needed to accurately perform test execution and logging, initial architectural spikes were performed. The developers of the main log tool had implemented the Bridge interface according to the requirements of the test department at EMP, but the interface was neither tested and the performance and possibilities were not explored before this thesis work was performed.

The existing architecture can be described as a collection of communication modules, transferring TVP and PrintServer data over a special data link protocol (DebugMux) and transferring this information to the PC over USB. Once the TVP packets and PrintServer data arrives at the PC, the DebugMux client (the main log tool) unpacks, parses and displays the information on the screen. The architecture can be described through the following picture:

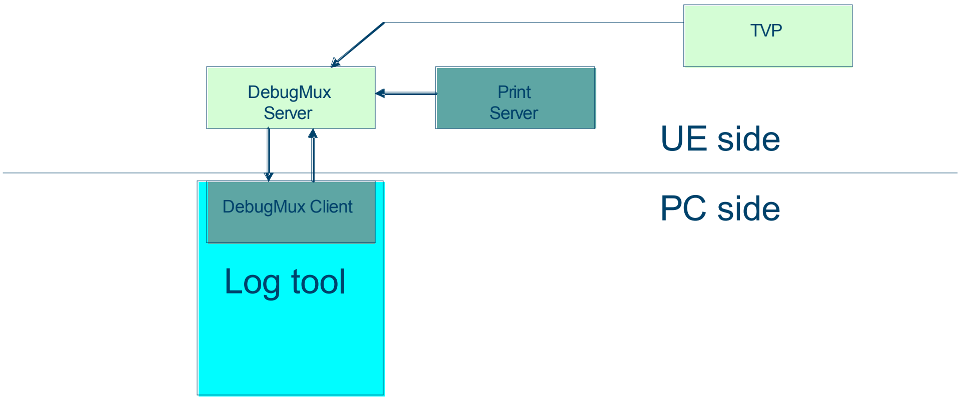

Figure 8: The existing log architecture

To make it possible to fulfill the new requirements, the Bridge interface was added to the main log tool (by the third-party developers), and this was later used in the practical work:

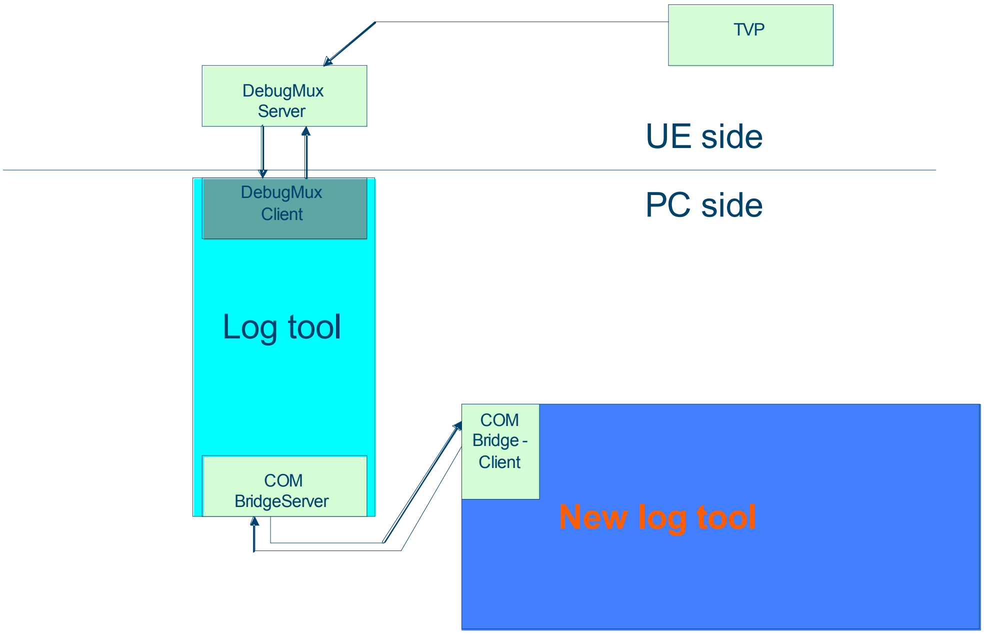

Figure 9: Architecture with support for Bridge

The Bridge interface is the prerequisite for all the practical work performed in this thesis. This existing architecture, together with the ideas presented in chapter 6.1, led to architectural spikes to decide what the most appropriate design would be. After having worked for a while with protocol stacks, the “layers” metaphor was chosen for the architecture – splitting the new log tool into a “COM communication”, “Data parsing” and “Presentation” layer.

One advantage of separating the inter-process communication layer and the presentation layer is that this makes adaptation against a possible future log tool interface is made much easier, if the decision is made to use another main log tool but keep the TVP protocol for logging. During work, this became a more and more attractive option, since the issues leading to drawbacks (described later) would then be more observable and controllable. The presentation primitives described in chapter 6.1 are all taken care of by the "Presentation" layer, while the control and triggering of log points is being taken care of in the other layers.

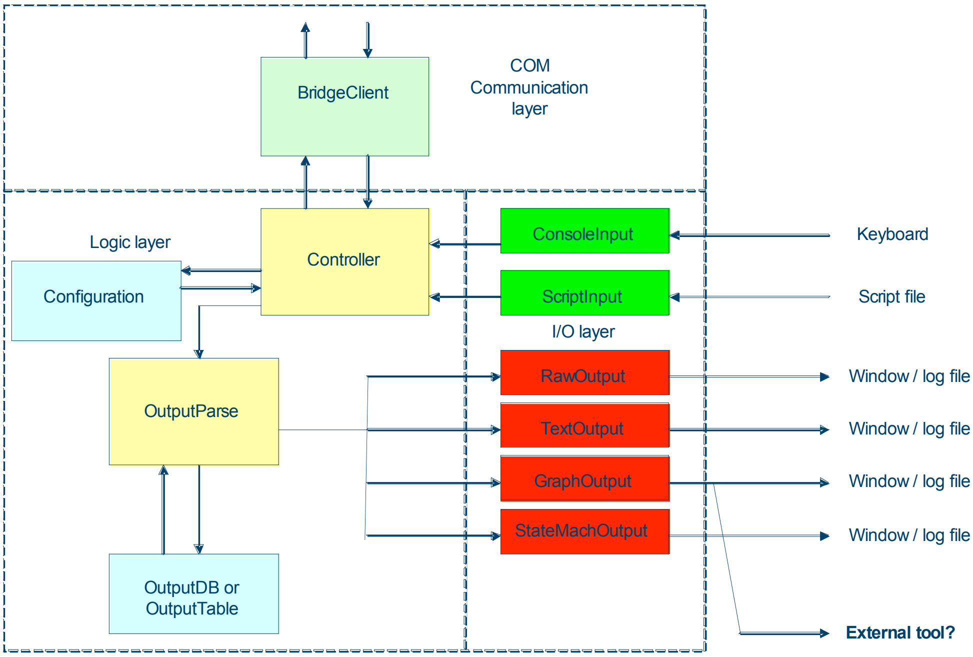

Figure 10: Architecture for the new log tool that fits with the existing interface

In the figure, the "COM Communication layer" could be modified in the future to connect directly to the "DebugMux Server" seen in figure 9, rather than going through the existing main log tool, but the other layers could be kept as-is.

The architecture contains several tables or information storages. For instance, the configuration when the tool is started needs to be read from somewhere and stored in a table. This can be implemented either by using a regular database, but for the architecture in figure 10, simple hash map tables (meaning that values could be accessed in O(1) time) were used. The output database has been created trough the use of "decode"-files, which guide how each log point (represented by a number) should be interpreted.

6.3 Implementation issues

After having worked out requirements and design, a lot of work went into attempting to realize the architecture. Here, unexpected problems related to methodology occurred, namely that the language choice was more important than initially thought for this type of inter-process communication in a Windows environment – Python does not seem to be mature enough for this yet. However, the architectural idea still meant that the situation could be remedied by separating control and communication into a C++ program, while performing parsing and presentation in Python, and then connecting the two prototype programs. The initial plan was to do everything in Python, since the total time spent coding could then be kept down.

The first attempts were made to construct all prototypes in Python, that is, including the COM interface implementation and everything. This was the first iteration, which almost succeeded – but Python, despite its flexibility and power, still lacked adequate COM interface support. COM interface support existed, but the COM interface provided by the main log tool investigated in this thesis was somewhat more advanced than ordinary COM servers and interfaces. This meant that the 100% Python solution was abandoned, after having seen it almost work together with the interface (all functions except for one could be activated, sadly this function was critical to actually receiving log data). Several variants of third-party Python modules with COM support were tested, including attempting to reverse-engineer the COM object and accessing the interface that way – something which actually worked with other COM interfaces, but failed for the Bridge interface.

Because of the lessons learnt during Python COM programming, the Java part of this thesis became rather brief – rather soon, the same problems as with Python regarding insufficient support for COM became apparent, and so this thread was also abandoned. The same kind of third-party modules as with Python were used, and they finally showed the same kind of drawbacks. This also meant that an Eclipse spike, which would have integrated parts of the new log tool into the main IDE, was put on hold as the Java modules did not fit with the COM interface.

The final solution was then to perform some of the experiments in C++, regarding the main communication through COM servers and COM clients, and to perform all other experiments in Python through shared data with this C++ application. This meant that some layers of the application are written in C++, while some layers are written in Python. The main drawbacks of this way of working is that performance is not as good as it could be – however, the main goal of this thesis is to show if the requested functionality is at all possible to achieve, performance issues are secondary when it comes to end application performance. Also, C++-programming can be more time-consuming that Python programming, and this thesis was meant to contain more experimental types of programming – still, this setback was overcome during later work.

After having used a very basic existing C++ prototype, the process of adding functionality and testing log point functions through the interface went rather smoothly – and this means that anyone wishing to continue this project will probably not face too many difficulties regarding the actual implementation, as the ground work has already been performed.

6.4 Tests of the prototypes

Moving from PrintServer logging to TVP logging in an external log tool is a big step for the testers. This means that quite some work needs to be done to ensure that a future new log tool is robust and usable in the same context as the old log tool. This is not something that was considered for the prototypes worked out here, but something which must be carefully tested before actual rollout (see “discussion” chapter). Keep in mind that for some test cases, especially during development of new functionality, a lot of data needs to be processed fast to be able to accurately debug the behavior of the mobile platform. This meant that fast and large log points were the main focus of the prototype testing, to see if the data being transferred over the Bridge interface arrived in the correct order and with the correct information in time.

Practical tests showed that the packets being transferred through the interface did arrive sequentially and with the correct data, but there was one major drawback – the entire TVP packet was in fact not transferred, parts of the header (containing the internal timestamp for when the data was generated) was not transferred. This drawback means that the packets must be timestamped when they arrive at the PC side, which is far too inaccurate to be usable. Investigations also showed that the DebugMux data link layer did not transfer packets all the time, but rather buffered the packets and sent them all at once every 200 ms (or when the DebugMux data buffer on the UE side was full). Since a WCDMA frame is 10 ms, and lots of information needs to be debugged on a frame-by-frame basis, this is not good enough for practical use. On the other hand, some log points on the PHY layer contains “internal” time stamps based on frame numbers (10 ms resolution), and for those log points it is possible to keep track of time. Still, any interface (including the Bridge) that relays TVP packets must not strip this vital information.

Apart from that major drawback, the functional tests went fine – a lot of different packets were activated and deactivated, and the information received was correct. For testing, actually existing log points were used, so the output from the prototype could be compared with the already existing representation shown in the main log tool[3].

[3] Note that the main reason behind the prototype is not to simply recreate what is already implemented in the main log tool, but to make it possible to add new log points rapidly and not have to wait for rebuilds of the main log tool to observe the same data – especially important when time is limited when new software is implemented in the UE.

It seems like the TVP module in the UE can put different kinds of timestamps (with different resolution) on the TVP packets, but that this is actually not part of the TVP packet that is sent through the Bridge interface. When disassembling a TVP packet, this is how it then looks in principle:

From UE to log tool: [Timestamp] + [TVP header] + [TVP data]
From log tool to Bridge interface: [TVP header] + [TVP data]

One reason for this behavior could be that the timestamp can be varied within the UE, while the TVP header and TVP data are created in a different way. Other functional tests involved designing and testing log point representations with the help of tables – plain text files were used for this, but in order to make things more generic, XML files should be used instead. The input required to make these kinds of tests were the log point database, where already implemented log points could be picked and tested, together with the log point specifications. These two are combined into a representation-textfile that guides to which output device the binary data should be piped, and how this should be done.

The process of testing new log points can be set up as follows:

- Decide which log point should be activated (for instance, “log point #400: RRC measurement reports”).
- Decide how to represent this log point (raw/print/graph/state/table), both tester and developer should collaborate.
- Write a new “decode”-file for this log point.
- Observe if the log point is periodic or event-triggered.
- Observe throughput and latency for this log point stand-alone or together with other log points.
- Observe timestamp and other information if available, compare to print strings to see that the correct information is transferred over TVP.

Several of the tests were made with different kinds of measurement information. The actual procedure for WCDMA measurement reporting is very complex, taking up a large section of the 25.331 standard [8]. Still, this type of more complex log point (meaning that the structure is dynamic, and the amount of information elements can vary) was still possible to represent in different ways without too much trouble by following the process above.

6.5 Maximum practical TVP throughput

Of course, it would be interesting to compare maximum TVP throughput with the throughput that can be seen using PrintServer logging. The PrintServer is constantly evolving, and works in a kind of unacknowledged mode – the PrintServer does not care whether the print strings arrive at the host, and if too many modules want to print at the same time, the buffer in the UE will be filled and prints will be discarded. This is not the case for TVP packets, where packets are sent acknowledged. At some data rate, however, it will not be possible to send enough TVP data without losing or severely delaying packets.

The exact measurements are confidential, but the important fact is that throughput per se is not the limiting factor. Breaking down throughput variables over the different layers, it can be seen that USB is not currently (at around 12Mbit/s theoretical, ~5Mbit/s practical) the bottleneck. The DebugMux protocol can also be tweaked to allow higher throughput. Rather, it is the processing power of the UE that sets the limit of how much information can be sent. For instance, if the UE is downloading data at a rate of 384 kbit/s, the CPU is already strained – if log information is added at different levels, the log data throughput would be approximately multiples of 384 kbit/s, but the practical limit is set by the UE processor.

6.6 Number of simultaneous TVP log points

An important consideration is the issue of how many log points could be activated at any one time. The processing of log points is more complex than simply concatenating everything into one print window, as with PrintServer logging. Testing the number of simultaneous TVP log points shown at one time is a good test of the parsing and representation parts of the prototype, as well as stressing the communication part (at both the UE and PC). Ideally, a large amount of log points could be enabled at any one time, to log many layers simultaneously. For specialized tests, it may be more important to focus on just a few, rapidly reporting log points (such as frame data, which is generated and sent every 10 ms). In practice, if the test case and the tester only needs certain log data, it seems to be good to not enable more log points than is needed. This is also something which should be controlled automatically, for instance through the test environment (when test case X is executed, load log tool configuration file associated with that test case).

The delay experienced when adding more and more log points is significant. This is also observed in the main log tool, which means this is not a bottleneck problem with the Bridge interface. When more and more log points are added, the UE response to adding further log points becomes more and more severe. This means that triggers and dynamic activation/deactivation of log points work much worse when there are already many active log points available in the system – not on the host PC side of things, since the tables and parsing is an efficient and well-known operation, but rather on the UE side of things or in the main log tool.

6.7 GUI and representation

The two previous chapters verify that TVP can be sent and parsed by the application, which covers the communication and data parse layers of the application. Now, in order to realize the advantages of the new log format, some representation needs to be tested. For this purpose, experiments were made with a GUI module for Python, called wxPython [10]. The advantage of this is that conceptual ideas can be tested, and then converted to another programming language (such as C++) if the concepts are viable.

Three of the output primitives outlined in the “requirements” chapter were suitable for GUI representation. State machines, tables and graphs are all good candidates for this kind of treatment. Graphs seemed to be the most difficult primitive to implement, but are nevertheless very important to the representation of radio logging. In order to simply test GUI operations, state machines were chosen as it was estimated to be faster to implement, and would still demonstrate how the entire chain, from log point generation to log point representation, would work.

Adding this kind of GUI to an application means that more concern must be given to things like threading and real-time processing of data, compared to before. However, no important issues were found when testing different kinds of GUI primitives – although it is time-consuming work, which is better left to the actual implementation phase of the real log tool. Note that the output tables described in 6.4 are used here as well – and aid from the developer is needed to realize how to best use the GUI.

6.8 Stimuli activation and triggers

The Bridge interface includes the capability to execute code in the platform through a process known as “stimulation”. This is something which can be used to trigger events based upon the reception of log points. One of the wishes was to be able to trigger code in the UE depending on what happens in the UE, which would be good for error handling, error recovery, etc. This was harder to implement using the interface, as a lot of internal information about the data structures was needed. It was discovered that stimuli activation and that kind of manipulation is something that is not at all used thus far by NS stack test, but several scenarios could be thought up where it would be useful. This meant that a way to trigger code execution within the UE from the regular software build is something which has not been used before, and in order to make it work, support for this feature (called “target test code”) was somewhat rough around the edges in the Bridge interface. Execution depended on knowing the exact layout of the data structures, and in the end only a few isolated experiments were performed.

Still, the use of stimuli is intriguing and certainly something that needs to be implemented in future tools. Since the platform is a complete OS in itself, it would be good to be able to trigger code in the platform at the same time the applications and real-time software modules were running – for instance, to check memory usage, to turn on/off functionality, and so on. This can be done using the “interactive debug” facility in the UE today, but the output consists of print strings only, not the desired binary data. Stimuli is a much more “interactive” way of debugging the platform, as the output is more flexible.

The format for using stimuli reminds of the format used for the actual TVP log points. As such, it fits logically within the log framework, and it will be possible to build scripts based on triggers and stimulation in the future. Already, Python scripts are used within EMP for various tasks, such as test case automation. Expanding this environment to also include TVP log point triggering and stimulation is a task very well worth undertaking. In chapter 4.2, some of the available options for log and control of the UE are shown – but the thing that is actually lacking is a good way of changing pieces of the UE software while the test case is running, something which is quite important for stress tests and load tests (and other system test cases as well), where some functionality should be exhaustively tested by switching it on and off, changing parameters, etc. This is possible using TVP stimuli.

6.9 Sampling and exporting log data

A sample-and-process feature was added to the prototype to show how it could be used in practice. This involves focusing on one single log point, but can easily be expanded to several log points. The log points are collected during a limited amount of time (such as 60 seconds), and the data is then fed into a regular application for further processing. Since other parts of the thesis work involved COM communication, it was a natural step to expand this thinking to involve other third-party applications. In this case, Python could be used for the COM communication, as the common applications used for demonstration purposes here have been Excel and Matlab, both of which implement COM interfaces.

Sampling in this context is not a real-time task, but still a good way to visualize what is going on for a few log points. This could be a good way to improve EMP logging functionality for customers – today, if a customer using the EMP platform needs to test or debug errors on his own, the instructions needed to create logs with the desired debug information can be cryptic, and the resulting log hard to understand. If sampling and processing log data can provide a clearer view of what is going on for the certain data points that need to be logged and debugged, which could aid the customer in recognizing what is actually being debugged. Sample-and-process also means that the presentation layer can be made much simpler than if the wxPython module is used.

7 Discussion of results and future work

The results above show that some important work remains to be done. First of all, the main log tool needs to be improved in two major areas. A minor point that needs to be added is that the timestamp must be preserved through the Bridge interface. This should be relatively easy to implement. The second point has to do with overall performance and latency of the log points. In order to use the interface in the desired way, the performance must be improved at several levels – investigations have shown that most of the latency reside in the main log tool application. Also, the resolution of the DebugMux protocol – sending packets only every 200 ms from the UE to the application is not enough to allow real-time triggers.

The combination of these two issues unfortunately make things hard for the user – it could be possible to compensate for one of these disadvantages. For instance, if timestamps were missing but log points were processed in real-time, the PC could stamp (less accurate) timestamps on the packets as they arrived. Also, if the timestamps were preserved, the latency drawbacks would not matter as much, since post-processing the logs would still work perfectly. Adding a timestamp to the Bridge output should not be very difficult, but will require changes to the implementation of the Bridge interface.

The application prototypes that are presented in this thesis will still work without modifications once these issues are resolved. The option to use a simpler, more light-weight interface instead of the main log tool should definitely be implemented – this became more and more evident during the experiments, as modifications to the prototypes could not overcome the shortcomings of the main log tool. The main log tool is far too large and too complex to break down and analyze from the outside. For instance, the communication between the main log tool and the Bridge middleware software is a COM interface of its own, which was not at all studied during the thesis work.

This option would require some work, but gives much more control over performance-critical parameters (both DebugMux and other protocol layers). It would require a rewrite of the basic Communication layer presented in this thesis, but it will not be a major task. Some of the major tasks needed to execute this is to solve the problems of authentication (a service provided by the main log tool together with the TVP protocol in the UE), DebugMux performance tuning and relaying data through the Bridge COM interface.

Presentation and data parsing, which has been tested using Python, could remain in Python form for the time being. In a real life application (meant for all groups at EMP and customers as well), the implementation should be rewritten in C++ for better performance and better presentation capabilities, especially if more extensive logging is activated. This is especially true for the GUI components, as the Python modules used for representation were not developed to be failsafe, and could not handle large amounts of data in a good way. Rewriting the Python modules to C++ means a few difficulties, mainly by converting the GUI code into MFC classes, but the pure programming tasks such as communication and data parsing can be converted on a class-by-class basis. Python as a language can be viewed as a C/C++ front-end, which means that the objects and structures are very much similar.

How much effort would be required to transform this thesis into something which could be used in EMP projects? Many parts could be kept as-is, but some refactoring and restructuring would be in order to complete the representation classes. Also, with the proposed table lookup and log point presentation solution, each new log point needs to be defined individually in the log tool. An estimation of the work needed to define each log point could be around 0-8 hours – depending on the complexity of the log point. Some log points can simply be printed, this is especially true for static log points that are proposed to replace static print strings – if a string is, on average, 40 bytes long, replacing each such string with a single TVP point containing a 16-bit index variable would save a lot of space. Other complex log points, that represent complex data structures and measurements, need some tuning to be able to represent (through graphs, state machines or tables). If a complete state machine needs to be drawn and the transitions mapped, it could take a full working day to accomplish this.

Taking the results and prototypes in this thesis and building a base application on this should take around 40 hours. Data parsing and table lookup should take around 80 hours to implement and test, and finally, basic representation classes should take around 40 hours to develop. Assuming 160 hours for the development of this C++ tool for the communication, data parsing and representation modules, the remaining part would be to also add as many log points as is feasible. Implementing and verifying a representation for each log point is a one-time effort, and maintenance of this work would not take any significant amount of time. The log point “representation database” could be stored in a version-controlled repository, and whenever the format of a log point changes, the developers and testers with a special responsibility for this functionality could also in parallel change the representation in the database.

This investment would probably be paid off, as less time would be spent debugging test issues. One huge improvement would be that there would be much less time spent adding and removing prints, rebuilding software and post-processing logs. Since this is work that involves 100-200 people in each project, the individual time savings might not be large, but the total time saved would be significant. Also, gradually switching to TVP would reduce processor load and memory usage for PrintServer strings in the UE if these are phased out. A gradual roll-out needs to be done, first by completing the implementation and internal testing work, then by letting some testers use the new tool in a regular product while other testers use the traditional way of logging. If the new log tool provides significant advantages for testability and tester productivity, and there are no adverse side effects in the UE, it would then be recommended to let all testers use the new tool in the following software project. Since each software project has a lifetime of roughly six months where test activities are performed, this means that a pilot project could be used during the next software project (where a limited amount of testers use the new tools), and after an evaluation, the following six months could possibly have a product variant where the PrintServer logging is disabled altogether.

8 Conclusions

In general, designing a log framework that lasts through many projects with different hardware and functionality is difficult. For this thesis, constraints have been placed by the logging architecture already in place, and the main log tool being used by EMP today (where more and more testers and designers will be involved, not to mention customers). This means that these constraints have, for the duration of the thesis work, been constant. Taken as a whole, the current architecture is partly suitable for making new and more efficient log tools, but most importantly, there are major drawbacks that limit any future tool functionality. The most important drawback is that the latency and throughput using current methods are not good enough to last for much longer. When adding more log points and increasing the rate of log data generation, the main log tool quickly becomes the bottleneck, consuming a lot of available memory in the PC and delaying arrival of log data. This means that one of the goals of the thesis, being able to design an adaptive log tool that can react to incoming log data by for example adding more logs or executing code on the target UE, is not possible for the moment. As the maximum tolerable latency for some log data is in the order of milliseconds, and the observed latency has been shown to be many seconds, this part of the thesis could be made to work with a better main log tool, but not in the current implementation. The delays have been roughly shown to exist in the main log tool, with some additional delays also occurring due to the DebugMux protocol implementation, as well as the overhead created by the COM server data transfers – many log points generate very little data, but generate data in such a rapid pace that the total processing time, compared to the amount of data processed, increases to high levels.

As for observability and testability, without regard to latency, the solution described in the thesis works good. Although extra loads with a higher rate of log data generation slows things down considerably, the implementation is simply a prototype written in a high-level script language, and it is still fast enough to provide a feedback which is more informative and easier to use for both testers, designers and customers compared to the current situation. Already, raw probe data can be used for debug information where a complete rebuild of the target UE software was needed. Besides, generating a good representation of the available log points (not part of the scope of this thesis apart from a few examples) is a one-time effort that will surely boost productivity as ambiguous results are less likely, and training time will also decrease as a result when the output of the actions taken are visible in a more concrete way. There is no doubt that the NS Stack Test department will be able to migrate to using binary log data, based on the experiments I have been conducting, as long as the supporting tools and interfaces are tuned to allow for higher throughput and less latency.

The ability to invoke code in the target UE based upon incoming log information also works, but the interface makes it a bit harder to implement this in a future log tool. However, it is a promising theme for future studies, and can be integrated with existing tools to make automated testing less of a burden to maintain. The possibilities to invoke target test code regardless of which software build is used in the UE is very intriguing, and developing a test environment based on TVP activation/deactivation/stimulation should be analyzed more thoroughly, perhaps by a new thesis.

Finally, the formal interface specification is good enough to build future tools upon, and is the most sensible step to take when developing the next generation of mobile platforms. Some care must be taken when attempting this migration, so this is preferably done as a pilot project, not involving all testers at once.

Appendix A: Abbreviations and terminology

The following abbreviations and special expressions are used extensively in the thesis, and basic knowledge of what they mean will make it easier to understand the text.

- 3G – The third generation of cellular mobile systems.

- 3GPP – The third generation partnership project, responsible for the specifications of certain 3G standards (such as WCDMA), and also responsible for future GSM/GPRS/EDGE specification evolution.

- ARM – A type of CPU for embedded devices, used extensively in most mobile phones.

- ASIC – Application Specific Integrated Circuit, for special-purpose chips, such as layer 1 WCDMA radio signaling.

- CDMA – Code Division Multiple Access, a way of separating users in a cellular mobile system through the means of different scrambling codes for different users.

- COM – Common Object Model, a standard way for inter-process communication in a Windows environment.

- DebugMux – The data link layer protocol used for communicating between the EMP platform and the host PC for log data.

- EDGE – Enhanced Data rates for GSM Evolution, a “2.75G” technology for improving data rates.

- EMP – Ericsson Mobile Platforms.

- GSM – Global System for Mobile Communication, the dominant “2G” standard employed around the world.

- GPRS – General Packet Radio Services, an evolution from GSM designed for packet-switched data.

- IOT – Inter-operability testing, a name for the kind of testing where the different parts of the system are tested together, for instance an Ericsson platform together with a Nokia radio access network.

- MAC – Medium Access Control, a layer 2 radio protocol for controlling when to access the air interface.

- NS – Network signaling, the most important protocol stack of a mobile phone.

- NS Stack Test – The department within Network Signaling that deals with lab testing, type approval and development test of new NS features

- OSI – Open Systems Interconnection, a model for representing communication protocols.

- PHY – Shorthand for “PHYsical layer”, or layer 1 in the OSI model.

- RLC – Radio Link Control, a layer 2 radio protocol for WCDMA.

- RRC – Radio Resource Control, a layer 3 radio protocol for WCDMA.

- RS232 – An interface used over a serial bus (“COM port”) for transferring data.

- TDMA – Time Division Multiple Access, a way of separating users in a cellular mobile system through reservation of timeslots.

- TVP – Test and verification protocol, a proprietary EMP protocol for handling log data.

- USB – Universal Serial Bus, the standard way of transferring data to and from a PC from small embedded devices, such as mice, keyboards, telephones, etc.

- UE – User Equipment, a 3GPP term for a mobile phone and a SIM card supporting 3G technology.

- U_u – The standardized name for the interface between the UE and the network.

- WCDMA – Wideband Code Division Multiple Access, a radio access technology for third-generation mobile networks.

- XP – eXtreme Programming, a light-weight methodology for software development.

Appendix B: References

1. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995

2. http://www.ericsson.com/products/mobile_platforms/

3. H. Kaaranen, A. Ahtiainen, L. Laitinen, S. Naghian, V. Niemi, *UMTS Networks – Architecture, Mobility and Services*, Wiley, 2005

4. http://www.eclipse.org/

5. http://www.python.org/

6. http://www.extremeprogramming.org/

7. Schiller, *Mobile Communications*, Addison-Wesley, 2003

8. TSG RAN Working Group 2 (Layer 2, 3), *25.331: RRC Protocol Specification 3.18.0*, 3GPP, 2004

9. http://www.daimi.au.dk/~datpete/COT/COM_SPEC/html/com_spec.html

10. http://www.wxpython.org/