INVESTIGATING METHODS FOR MEASURING NETWORK CONVERGENCE TIMES

Jakob Danielsson
jdn11003@student.mdh.se

Tobias Andersson
tan10006@student.mdh.se

Examiner: Thomas Nolte
Mälardalen University, Västerås, Sweden

Supervisor: Mohammad Ashjaei
Mälardalen University, Västerås, Sweden

Company supervisor: Thomas Sörensen,
Westermo, Västerås, Vallby Institutet
June 10, 2016
# Table of Contents

1 Introduction ................................. 5
   1.1 Motivation .................................. 6
   1.2 Thesis outline ............................ 6

2 Background ................................. 7
   2.1 The OSI model ............................ 7
   2.2 Convergence time in layer 2 .............. 7
      2.2.1 FRNT - Fast Reconfiguration of Network Topology ........................... 8
      2.2.2 FRNT Ring Coupling ....................... 8
      2.2.3 Multi-Link Dual Homing .................. 8
      2.2.4 Rapid spanning tree protocol - RSTP and STP .......................... 8
   2.3 Convergence time in layer 3 .............. 9
      2.3.1 RIP - Routing Information Protocol .......... 9
      2.3.2 OSPF - Open Shortest Path First ........... 9
      2.3.3 VRRP - Virtual router redundancy protocol .................. 9
   2.4 Convergence measurement tools ............. 10
   2.5 The Linux operating system ................. 10

3 Related Work ............................... 11

4 Problem Formulation ......................... 13
   4.1 Goals .................................... 13
   4.2 Research questions ........................ 13

5 Method .................................... 14
   5.1 Open traffic measurement tools ............. 14
   5.2 Real-time operating systems ............... 15
   5.3 Test program on real-time operating system .. 16
   5.4 Testing equipment .......................... 16
   5.5 Testing suite ................................ 17
   5.6 Bare metal programming ..................... 17
      5.6.1 Programming a Raspberry Pi ............ 17
      5.6.2 Arduino based solution .................. 18
      5.6.3 EVK1100 based solution .................. 18

6 Traffic engine development .................. 19
   6.1 FreeRTOS network stack ..................... 19
   6.2 Bare metal Raspberry Pi .................... 19
   6.3 Arduino UNO with Ethernet shield W5100 ...... 20
   6.4 EVK1100 .................................. 20
   6.5 Packet design ................................ 20
   6.6 Analyzing a packet stream .................. 21

7 Comparison study ........................... 23
   7.1 Real-time operating systems ............... 23
      7.1.1 Schedulers ............................. 23
      7.1.2 Memory management ...................... 23
      7.1.3 Preemption protocols ................... 24
      7.1.4 Documentation .......................... 24
   7.2 Open source packet analyzer tools .......... 25
      7.2.1 Kernel based tools ..................... 25
      7.2.2 User space tools ....................... 25
   7.3 Evaluation summary ........................ 25
   7.4 Optimizing user space tools ............... 26
   7.5 Traffic engine test cases ................... 26
7.6 Performance test 1 ........................................... 27
7.7 Performance test 2 ........................................... 27
7.8 Reliability test 1 ........................................... 27
7.9 Reliability test 2 ........................................... 29
7.10 Reliability test 3 ........................................... 31
7.11 Usability evaluation ........................................ 31
7.12 Network performance tests ................................ 33
7.13 Measuring network convergence ......................... 33
   7.13.1 Packet losses ........................................ 34
   7.13.2 Protocol jitter measurement ........................ 34
   7.13.3 Protocol worst case convergence time ............. 35
8 Results & Conclusion ........................................... 39
  8.1 Summary of results ....................................... 39
     8.1.1 Automated test system .............................. 39
  8.2 Conclusion ............................................... 40
  8.3 Limitations ............................................... 42
  8.4 Future work .............................................. 42
A Appendix ....................................................... 46
  A.1 FreeRTOS and USPI setup ............................... 46
  A.2 PKTgen setup ............................................. 46
  A.3 Running the TShark scripts ............................. 48
  A.4 Configurations used in the switches .................... 48
     A.4.1 FRNT configurations ................................. 48
     A.4.2 RSTP configurations ................................. 49
     A.4.3 RIP configurations .................................. 50
     A.4.4 OSPF configurations ................................ 52
  A.5 Test results .............................................. 53
Acknowledgments

This thesis was done in collaboration with the R&D department of Westermo teleindustri AB located in Vasteras. We would like to direct a special thanks to our company supervisor Thomas Sorensen who helped us configure the switches and routers as well as providing valuable inputs on how to design our test system. We would also like to thank our university supervisor Mohammad Ashjaei who provided support throughout the entire thesis.
Abstract

This thesis investigates different methods that can be used for analyzing network performance and, ultimately, can be used for measuring the convergence time of ring coupled networks. As of today, many networks are often run with extra links, serving as backup links in case any of the main links would go down. To operate networks with backup links in layer 2 and layer 3, specific rerouting protocols such as RIP and OSPF are used in order to calculate a feasible path through a network when a network state changes. Depending on different implementations of the protocols and the hardware used, the convergence times can vary substantially, which means measuring the network performance is a very important part when developing a network solution. To execute network tests, a packet engine suite is used consisting of a network traffic generator that is used for creating a packet stream, as well as a traffic receiver that fetches the packets sent. Various types of engines can be used including Linux based, real-time operating systems based and bare-metal based solutions. From these different types of engines, a few tools are chosen and investigated on different properties including performance and usability. It was found that Tshark (Linux, RT-Linux based), USPI (Raspberry Pi bare metal), FreeRTOS (Raspberry Pi based), Arduino and PKTgen (Linux kernel based) were the most suitable approaches to be used for testing. The test parameters include testing the gaps between packets, maximum jitter, average jitter and packets sent per second. These tests revealed that an IXIA solution was slightly more accurate when used as a receiving end since it produced less jitter, however this difference could only be noticed in a micro second range. It was also revealed that it produced slightly less jitter than the other packet generators, also here only noticeable in a microsecond range. Thus it can be concluded that IXIA is not much superior any of the close to hardware solutions. The executed network tests revealed that the Westermo developed layer 2 protocol FRNT generated less network convergence time and less packet losses than the commonly used RSTP protocol. Similar tests against the layer 3 protocols revealed that RIP was much faster than OSPF and it also lost less packets. Finally it is concluded that there is no need to buy an expensive network testing suite to test the convergence time of a network. Instead, a network testing suite can be developed with minimal funding.
1 Introduction

The networks used in today’s industry often involve complex switching and routing algorithms due to the size of the network. When a link goes down in a network, which may be due to cable errors or software errors, the network must re-route. This means to "converge" into another feasible route through the network. This is a process which can occur in both the switching layer (layer 2) and routing layer (layer 3) of the OSI model. Testing the performance of the convergence process is a very important task, since it reveals how well a protocol performs. Nowadays, tools for measuring the convergence time exist, being commercially developed programs or open-source developed programs. The commercial tools are expensive, whereas the open-source tools are not always quality guaranteed. This thesis will focus on evaluating different open-source solutions for measuring network convergence time. Typically, the simplest test scenario consists of sending measurement packets through a network with a sender connected to the starting node and a receiver connected to the ending node. By counting the packets lost during a link failure, it is possible to get the convergence time [20]. Much focus will be directed towards sending accurate test data, this in turn means that if a packet is supposed to be sent once every millisecond, it should do so without any deviations. This means that the test program must be run on a Real-time operating system, so that predictable sending intervals can be guaranteed. A typical test scenario in layer 2 can be seen in Figure 1 where a transmitting packet generator (Packet gen 1 in Figure 1) has been connected to the start link of the network and a receiving device has been connected to the end link of the network. When Packet gen 1 sends a packet into the network it must first go through the links that can be seen in the figure. In order to find a way through the network the used protocol must decide which route is most feasible.

![Figure 1: Layer 2 network example](image)

The second testing will aim towards testing the convergence routing protocols - "interior gateway protocols". By testing the routing layer, it is possible to perform a convergence test against an entire network. Figure 2 depicts an example of how a layer 3 architecture can look like. It also depicts how different packet generators can be used to test different parts of the subsystems. The example depicted in Figure 2 shows a network that messages must go through. In this example there are routers connected together instead of switches, and there is more than one packet generator. However the principle remains the same, which means the protocol being used by the system will determine the best path from the packet generator to the Packet rx node.

An important mention regarding a solution which can be connected to an existing network is the clock synchronization issues. Since the traffic analyzer and receiver programs run on different devices, one must be aware of the units clocks, they may not be synchronized in the end of a test.
This synchronization fault may cause errors when calculating the packet transmission time through the network which in turn may lead to inaccurate jitter measurements. There may exist a need to implement a clock synchronization protocol, depending on the drifts between the clocks.

1.1 Motivation

Currently, one of the few solutions available for measuring network performance is created by the company IXIA. IXIA provide a real-time solution to the problem, however it is very expensive. This thesis will investigate to if there are any open source tools available that can be modified to provide similar results. These open source tools will be compared with IXIA’s own network tests, as well as tests developed specifically for this thesis.

1.2 Thesis outline

In this thesis we present a study of how it is possible to measure the convergence time in a reliable way by investigating the performance of various network analyzing tools. Chapter 2 provides background information. Chapter 3 presents related works to this thesis. Chapter 4 contains the problem formulation and the research questions presented in this thesis. Chapter 5 describes the methodology of how the testing should be done. In Chapter 6, the methodology and requirements to this thesis are explained. Chapter 7 shows a comparison study over different tools and operating systems. Finally, Chapter 8 explains the results and conclusions drawn from the thesis.
2 Background

Currently, Westermo Teleindustri AB [35] uses a test automation framework called Fawlty to verify and validate functional behavior of their operating system WeOS. However, non-functional testing such as performance and robustness tests, has not been fully automated, but has been done either by manually operated or semi-automated tests. This procedure has been proven to be difficult and time consuming. To solve these issues, non-functional tests regarding robustness and performance will be implemented in Westermo’s switching and routing network. The current Westermo devices uses network convergence protocols FRNT or RSTP to handle if a node goes down at layer 2. The devices also use the OSPF or RIP re-routing algorithms if a node goes down at layer 3. Westermo has a test method for the convergence time of a system, i.e. the time from which the link is disconnected until a new link is found, however their test method is not automated. This could be solved by using already existing network testing tools, however they are either not feasible because of their cost, or it is unknown if the tools provide accurate results. To be easy to use, the test system should be executed on an portable embedded device which must have a real-time capabilities. A predictable system is required for testing the downtime of the network, (RDI - routing dead interval) as too high jitter may alter the test results drastically.

2.1 The OSI model

The OSI model is known under the standard number ISO/IEC 7498. The OSI model is used for depicting a model of how computer communication work. The OSI model is divided into 7 different layers which are all independent from each other. This means that layer 3 is independent of the techniques which are being used in the under- or overlying layers. Figure 3 shows the entire OSI model.

![OSI Model](image)

Figure 3: The OSI model

As mentioned before, the layers which will be investigated in this thesis are layer 2 and layer 3 which are implemented into a Westermo network. Layer 2 is the switch layer and decides from which MAC address to which MAC a packet should be sent. By using different protocols in this layer, it is possible to decide which nodes a packet should travel to in order to find the destination point of the network. Layer 3 is the routing layer which decides a feasible route through a network, by using specific routing protocols, it is also possible to optimize the traveling route inside a network. The protocols which currently are supported by the WeOS are listed in Table 1:

2.2 Convergence time in layer 2

The most common device found in second network layer is a switch. The problems that can occur in this layer can have several effects on the network, such as long response time and worse
Table 1: A subset of protocols supported by WeOS

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRNT</td>
<td>2</td>
</tr>
<tr>
<td>STP &amp; RSTP</td>
<td>2</td>
</tr>
<tr>
<td>OSPF</td>
<td>3</td>
</tr>
<tr>
<td>RIP</td>
<td>3</td>
</tr>
</tbody>
</table>

performance. These problems can be caused by four different categories of problems. Firstly, capacity can be the reason, as there is a limit to how many MAC addresses that can be used. Secondly, overheating can also lead to these kind of problems. Thirdly, authentication failure. Finally, moving the hardware can also lead to trouble. If hardware is moved without any testing done afterwards it can be difficult to know if the system works as it should afterwards [20].

These previously mentioned errors can occur in this data link layer network. Because of these errors a new feasible path may need to be found through the network. The new path through network is decided by which protocol is being used. Some of these protocols, such as spanning tree, link OAM, and service OAM are used to fix problems that occur in layer 2. There are also some protocols that provide a redundancy in the layer 2 networks, such as STP, RSTP and MSTP. New protocols with less than 50 milliseconds time to converge are on their way according to the study presented in [20].

Westermo’s operating system WeOS has support for a few different protocols in layer 2, which are FRNT, FRNT Ring Coupling, and the two spanning tree protocols, RSTP and STP [36].

2.2.1 FRNT - Fast Reconfiguration of Network Topology

FRNT is a protocol that, as the name suggests, is made to reconfigure the network topology quickly. An FRNT configured network consists of ring connected switches, i.e. each switch has links to two other switches. One of these switches is called a focal point. This switch will block traffic in one direction. However, if a link failure happens, an event will trigger in the other nodes and they will send a message to this focal point to open the link, allowing traffic through this instead. Should the connection through the links return, then the focal point will be informed and it will once more block direction it had been previously blocking [36].

2.2.2 FRNT Ring Coupling

FRNT Ring Coupling allows two or more FRNT rings to bridge together. Each of these FRNT Ring Coupling nodes can have multiple uplinks with the other nodes. However, only one uplink will be active at a time. A FRNT Ring Coupling can be viewed as two different routers connecting two different Layer 2 networks, however Westermo recommends that the protocol should not be used as such [36].

2.2.3 Multi-Link Dual Homing

Multi-Link Dual Homing is quite similar to the FRNT Ring Coupling in that it allows another node to connect to a FRNT network. However, in multi-link dual homing this node is a single switch on its own. One of these nodes can have several uplinks to different FRNT nodes in the FRNT ring, but only one of these uplinks will be active [36].

2.2.4 Rapid spanning tree protocol - RSTP and STP

The spanning tree protocol builds up a switched network into a tree and allows access to certain nodes from another node through this tree structure. The ports are assigned a cost to find the shortest path through the network without any loops. With a given interval the switches will send hello messages to check if nearby nodes are still open or not in order to detect network failures. The difference between RSTP and STP is that RSTP is quicker, however it has a limit to the network size [36].
2.3 Convergence time in layer 3

The routing layer decides which path is most feasible through a network using a routing algorithm. If a link inside a node is down, the entire network has to be re-routed to another path. These paths may vary depending on which routing algorithm is used. The current routing algorithms which are used are called distance vector and link state algorithms [21]. When using a distance vector algorithm, a node only knows the distance to its nearest neighbours. By always calculating the minimum route to a node’s nearest neighbour, until the end destination has been reached, it is possible to create a full route through a network. While distance vector algorithm relies on what the node knows about everyone, link state algorithms depend upon the node’s knowledge of its neighbours. Whenever a link is found to be dead, the network will have to re-evaluate which is the best path through the network. WeOS supports the RIP and OSPF protocols. Figure 4 shows an example of how the system can converge using either distance vector or link state algorithms, whereas the distance vector algorithm chooses a route which as actually longer than the shortest path possible.

![Figure 4: Convergence example](image)

2.3.1 RIP - Routing Information Protocol

RIP is one of the oldest distance vector protocols and has been used since the early days of ARPANET [16]. The RIP protocol is based of the bellman-ford algorithm, also known as the distance vector algorithm which is used to calculate a path through a network. A big limitation of RIP is that it only allows for 15 hops, making it unsuitable for large networks.

2.3.2 OSPF - Open Shortest Path First

Open Shortest Path First is a link state algorithm, which uses Link State Advertisements(LSA) to advertise the status of the router. OSPF has 5 different types of messages which are being broadcasted to a node neighbours: Hello messages, Database description, Link state request, Link state update and Link state acknowledge. Due to the limitations in number of hops of RIP, OSPF is suitable for larger enterprise networks or when very fast convergence times are needed [4].

2.3.3 VRRP - Virtual router redundancy protocol

The primary objective of the virtual router redundancy protocol is to enable deployment of additional routers in an IP subnet as backup router in order to support redundancy. Like OSPF and
RIP, this algorithm can be used to converge the network. When the node goes down, the backup router will become the new master. VRRP enables the sharing of virtual IP addresses, which means a master router can have the same virtual IP as a backup router. In order to pick which one is the master, priorities are assigned from the gateway router. The highest priority becomes the master router, the others become backup routers [25].

2.4 Convergence measurement tools

As mentioned before, there exist both commercially available tools and open-source tools for measuring the convergence time. Commercial tools include IXIA [19], Various XENA [22] software. One of the main goals of this thesis is studying the performance and reliability of different open-source which can be used for testing layer 2 and 3 networks. The simplest tests which can be executed consist of a constant packet stream which is being sent from the start node to the end node. If the node has not reached its destination in a certain amount of time, a link is considered to be broken [20]. A full review of all tools to be tested and their properties are listed in Section 5.

2.5 The Linux operating system

The Linux operating system is a Unix-like operating system which is assembled under the open source software development and distribution. Linux is generally used as a general purpose OS, however there exist versions of Linux which support real-time performance such as Linux with RT preempt patch. General Linux is separated into two different modes, kernel space and user space. User space handles functions which are not time-critical functions such as functions from the C library API. Kernel space handles time-critical functions such as IO functions. The current default scheduler of Linux is the Completely Fair Scheduler (CFS) which uses the scheduling algorithm "weighted fair queuing" [24]. The CFS has three different scheduling policies, SCHED_NORMAL, which is the policy for a normal task, SCHED_BATCH, which does not preempt as often and SCHED_IDLE which is weaker than the lowest priority possible in Linux. There is a possibility to add round robin (RR) and first in first out (FIFO) in order to improve upon the real time behavior of Linux.
3 Related Work

There exist some work regarding the area of network convergence time (NTC) and real-time operating systems (RTOS) comparisons. Siddiqi and Nandy [28] discuss how it is possible to optimize the convergence time of the OSPF algorithm located in OSI layer 3 and also discuss the problems with sending hello messages together with normal messages which can cause congestions. To solve this problem, a decrease of the interval between hello and RDI messages is proposed, which will ensure that network failures are detected more quickly. To furthermore decrease congestion inside the network, a tunable OSPF protocol was created with less RDI. This shows the importance of taking congestion into account when testing the RDI. In order to optimize the convergence time of an OSPF-TE network, Basu and Riecke [2] suggest using two triggers for optimization: periodic triggering which refreshes traffic information that has reached its age limit and threshold based triggering. To measure the network convergence time, hello messages are sent in a subsecond range. It is concluded that this optimizes the timeout period of the hello messages significantly, however this increases the processor load. These reports offer valid points on how to optimize convergence systems, however not much about how the testing program is interacting with the protocol. Moreover, they are not suggesting which real-time aspects should be accounted for. Vidalenc et al. [33] worked on a study to make self-healing mechanism to improve convergence times in networks. The paper proposes a solution that is based on predicting when a failure will happen. They observe the routers health and make decisions based on reading information from the router. These readings can involve information such as the temperature of the unit and its power supply voltage. The results of this study show an improved availability, however at extreme conditions such as very high prediction rates the system showed some potential limitations.

Convergence networks are designed in different ways, which means they behave differently. Che and Cobley [5] published a comparison study on how voice over IP performs in different networks, link state, distance vector and hybrid protocols. The properties of these different networks (OSPF, RIPV1, EIGRP) are presented. Threshold parameter values are also presented which are needed to maintain an acceptable VoIP call. These parameters include delay, jitter, packet loss and MOS. The study was however not done as an actual implementation, instead they use a simulation tool called OPNET to simulate the network and the testing packets. The methodology of the testing were to send data streams at different bitrates for 30 minutes and by deliberately failing 2 links in the networks after 10 minutes it was also possible to measure the convergence time. The results given revealed many problems which can occur during network initialization, including jitter and congestion problems. It is also concluded that RIPV1 is not affected by the failed links, as RIPV1 chooses another route than OSPF and EIGRP. This is different in our work where we choose a realistic environment. As we will test in a real environment in a ring connected network, RIP will not be able to choose another route wherefore the convergence times can be measured.

Pham et al. [23] researched how to decrease rerouting time on mobile ad hoc networks. In the research it was discovered that a large part of the re-routing time is based on the queueing. The problem was simulated using the network simulator NS-2. It was found that the size of the queue increased the rerouting time greatly. The authors proposed a solution to the queue problem by making the retry limit adaptive. They seemed to completely remove the queueing problem once they put their solution into the simulations. When configuring the switches, it may be required to consider settings for queues in order to optimize the performance. Another work made on mobile ad hoc networks exists where an enhanced approach was tested. Hussain and Khader [18] implemented virtual zones in the buffer zones of the optimized link state routing protocol (OLSR). These virtual zones hold information of the neighbouring virtual zones. Their virtual zones have the benefit of improving the rerouting time, however they also create more overhead.

In order to get a high quality on the testing result, the testing platform must be reliable and accurate. Dezhi et al. [9] presents a work where the real-time operating system VxWorks 5.5 was used to handle the tasks which were run. It was concluded that the information gathered from these tests allowed them to find time consuming problems in the network. The focus on real-time operating system makes it clear how important real-time is when looking at performance tests. This shows how important it is to consider real-time operating systems in this thesis.

Due to this subject being very specific, not much directly related work can be found on how to compare real-time operating systems for networks. However, some studies have been made.
which compares the performance of different real-time operating system using various parameters. Hambarde, Varma, and Jha [15] made a survey over real time operating systems where different RTOS capabilities are compared including deadlock management, memory footprints, portability development tools provided, security and run time performance between different operating systems. The real time operating systems compared were VxWorks, Windows CE, QNX Neutrino and RTAI. The performance test compared the latency, which was found by measuring the time between triggering an interrupt and the device responding to it. The jitter was stated to be the random variation from multiple latency measurements. Lastly, the performance evaluation also includes the worst case response time of the different operating systems, which was found by analyzing the maximum interrupt frequency. It was concluded that RTAI is very suitable for smaller applications, while VxWorks is great for more demanding work. It may be worth considering their latency test when measuring the convergence time in this thesis in order to find a truly accurate measurement.

Tan and Tran Nguyen [31] made another survey focusing on both comparison for several different real-time operating systems and performance benchmarking on 4 different RTOS’s. Firstly, it is explained what an RTOS is and how it can be used for greater performance of software execution, due to time-predictable behaviour. This strengthens the argument for using an RTOS in a network convergence testing system. While an RTOS may not be crucial for execution of a network test, RTOS features such as API’s will make the development process much faster. When multiple networks are going to be tested, it may however be crucial to use an RTOS since multiple tasks will be executed simultaneously. Although comparison studies over different features of an RTOS is important, it does not quite fit in this scope. However using the benchmarking techniques presented by Tan and Tran Nguyen can be very useful. Nevertheless, not all tests are compatible with testing a network traffic analyzer, as task the amount of tasks active can vary for more than only 2 different tasks. Instead measurements have to be taken for an amount of tasks (where n is the amount of routes going to be tested) to see how the performance will change for interrupts. This thesis will also execute tasks which will use the actual Ethernet port, which may alter the end result of a task’s data, hence making it a reliability test as well. Finally Tan and Tran Nguyen conclude that there is no superior RTOS. The user instead has to look for specific requirements and chose an RTOS according to them.

Xiao, Wei, and Yungxiang [39] have made a system for monitoring the performance of a network. The parameters monitored were network throughput, the utilization rate, delay time and packet loss rate. The paper discussed the impact on the performance of the network that the network monitoring system would cause. Several modules built up the system. Each module had a job, and these were to sniff packets, analyze the protocols, control communication, generate packets, transfer packets, and one served as a rules library. A performance module for performance evaluation was added as well as a module for performance feedback. The conclusion of the paper was that these added modules allowed the system to dynamically adjust the impact from the network monitoring system on the network performance. It is useful to consider the possible negative sides that a traffic analyzer can have on the system. It would not be a good performance test on a network if the traffic analyzer would for example make the network lose packets, so this will have to be considered when testing and creating a traffic generator and analyzer. However the paper does not include consideration for real-time and jitter, which are important factors when measuring the network convergence time.
4 Problem Formulation

The objective for this thesis is to find and evaluate real-time network engines for Westermo Teleindustri AB and integrate these into their already existing test platform Fawlty. The network engines can run as either standalone applications, run under a regular operating system or a real-time operating system. The performance tests that are investigated in this thesis are tests with respect to network performance and robustness. These tests include creating high traffic network streams with the lowest jitter possible while still maintaining high precision and reliability.

4.1 Goals

The main goal of this thesis is to present an open source tool that will accurately measure the convergence time of a network. To reach this goal, the thesis addresses the following sub-goals:

- To provide a survey of a number of already existing open source tools.
- To provide a comparison of the results of a tool created during this thesis with the results from the open source tools.

4.2 Research questions

In this section, the research questions based on the presented goal are stated.

1. Different operating systems will be investigated and their properties regarding feasibility for testing convergence time in layer 2 and layer 3 networks. Which operating system is most favorable to use in this scenario?

2. The switching protocols use very different technologies, thus which protocol is most efficient?

3. Different routing protocols use different methods to converge, thus which routing protocol can be used most efficiently to reduce the convergence time of a network?

4. Since network congestion and jitter may occur in a network when testing the convergence time, a real-time measurement of the time is needed. How can we measure the convergence time in real-time?
5 Method

To answer the research questions provided in Section 4.2, different areas have to be investigated. These areas are listed below.

- Evaluate Open Source IP network performance test & measurement tools and appoint the most suitable ones. This evaluation will be done by investigating different parameters of the open source tools, such as supported protocols, supported layers, real-time capabilities and more.
- Evaluate Open Source Operating Systems and appoint the most suitable one as a platform for traffic engine testing.
- Install and set up a test system for performance testing. This point includes several sub-tasks with the goal of achieving a complete network system test.
  - Implement a performance test framework on a real-time capable unit.
  - Execute a performance test on a layer 2 subsystem environment.
  - Execute a performance test against a full layer 3 network.
  - Execute performance tests against different hardware.
  - Analyze the test results.
- Integrate the best framework with Westermo’s current test automation framework Fawlty
- Propose suggestions of improvements on non-functional testing.

5.1 Open traffic measurement tools

This chapter presents different tools related to this thesis. The combined traffic generator/traffic analyzer tool will have to fulfill the following requirements in order to be useful for calculating network convergence time in Westermo’s networks:

- General requirements
  - Command line interface
- Traffic generator capabilities
  - Number of packets to be transmitted
  - Jitter
  - Latency
  - Specify packet size
  - Be able to change the type of service field in the IP header
  - Support for multiple traffic flows
  - Delay between packets
- Traffic analyzer capabilities
  - Number of packets received
  - Number of packets lost
  - Jitter
  - Latency
  - Log data to file
Many packet generators exist today, therefore many were tested to evaluate their suitability. Hping3 and Mausezahn both provided useful options for a traffic generator. Ngrep and Tshark show the required information that is needed in the traffic analyzer and appeared reliable. TCPdump turns out to be too slow and could not keep up with the messages transmission, hence it was deemed not reliable enough. Iperf and Nethogs both lacked in the data which was displayed, as they did not provide the information required. Iperf could have potentially been a useful packet generator, however it needed its traffic analyzer as a server in order to start sending. This made Iperf not suitable as the analyzer part only showed the data rate. Table 3 shows the tools that were suitable.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Installation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hping3</td>
<td>Traffic generator</td>
<td>apt-get</td>
</tr>
<tr>
<td>Mausezahn</td>
<td>Traffic generator</td>
<td>apt-get</td>
</tr>
<tr>
<td>Ngrep</td>
<td>Traffic analyzer</td>
<td>apt-get</td>
</tr>
<tr>
<td>Tshark</td>
<td>Traffic analyzer</td>
<td>apt-get</td>
</tr>
<tr>
<td>TCPdump</td>
<td>Traffic analyzer</td>
<td>apt-get</td>
</tr>
<tr>
<td>Iptraf</td>
<td>Traffic analyzer</td>
<td>apt-get</td>
</tr>
<tr>
<td>Nethogs</td>
<td>Traffic analyzer</td>
<td>apt-get</td>
</tr>
<tr>
<td>Iperf</td>
<td>Both</td>
<td>apt-get</td>
</tr>
</tbody>
</table>

Table 2: User space tools that have been reviewed

It is important to mention that these tools are user-space implemented, which means they can be affected by the Linux kernel interrupts. Another important pointer regarding the traffic analyzer tools is that they only implement traffic "sniffing", i.e. running in promiscuous mode to be able to sniff the network card.

There also exist kernel based traffic analyzers, these tools being run as modules within the kernel. This will allow less kernel interrupts as it is possible to set priorities of these tasks. However, higher priority tasks will still be able to preempt these programs. Since the kernel based tools add more programming complexity, less open source kernel based tools have been developed. A list of the kernel based tools is presented in Table 4. Note that the tools are not limited to the mentioned ones in the table, however they are available to our requirements.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Installation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pktgen</td>
<td>Traffic gen/analyser</td>
<td>compile from source</td>
</tr>
<tr>
<td>KUTE</td>
<td>Traffic gen/analyser</td>
<td>compile from source</td>
</tr>
</tbody>
</table>

Table 3: User space tools suitable to the requirements

Table 4: Kernel based tools

5.2 Real-time operating systems

To be able to run a test from a real-time operating systems some requirements are needed from the RTOS to be able to execute a test in a reasonable time frame. Firstly, the RTOS must have support for a network stack. When testing layer 2 it is also crucial to be able to alter the source and target MAC address of a packet. It is also crucial to be able of setting source and destination IP of layer 3. To conclude, it is a value to have full control of layer 2 and layer 3 APIs. More restrictions also exist regarding the hardware architecture compatibility. The hardware platforms which are being
used for testing in this project is a Raspberry Pi model B2, an atmel EVK1100 board and Arduino Uno Revision 3. The Raspberry Pi uses an ARM architecture while the EVK1100 and Arduino both support an Atmel architecture. Lastly, the real-time operating system must be open-source. Table 5 shows different real-time operating systems that match the criteria.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeRTOS [3]</td>
<td>Modified GNU GPL</td>
</tr>
<tr>
<td>ChibiOS_RT [34]</td>
<td>Modified GNU GPL or proprietary</td>
</tr>
<tr>
<td>Contiki [7]</td>
<td>BSD license</td>
</tr>
<tr>
<td>VxWorks [37]</td>
<td>Proprietary</td>
</tr>
</tbody>
</table>

Table 5: Suitable RTOSes and their licences

While VxWorks is not open source, it will still be a part of the comparison because of its popularity.

5.3 Test program on real-time operating system

The test program created on a real-time operating system requires two tools to be implemented. For creating packet streams, a network traffic generator is needed. The traffic generator will be used to create packets which should be sent through the network. The payload of the packets will include a time stamp which is the time when the packet is sent from the device. This is a useful property since it can be used for measuring the travel time through the network. It will also include a sequence counter which can be used for checking if packets have been lost during the transmission.

The traffic generator will have to generate these packets in a specific bit rate which should be possible due to the use of a real-time operating system. The test system also need a traffic receiver which should sniff the network and receive packets from the traffic generator. The traffic receiver will be used for calculating the travel time of a packet and will be used for determining if a network has failed.

Since the test program will be implemented on both an AVR32 bit processor and an ARMv7, the firmware configuration will differ for enabling Ethernet access.

5.4 Testing equipment

The testing will be performed against the Westermo RedFox network device. It is possible to configure the RedFox as a switch and a router with OSPF, RIP, RSTP and FRNT capabilities. The RedFox device is depicted in Figure 5.

Figure 5: Redfox switch system
5.5 Testing suite

In order to execute a testing suite for a network convergence measurement solution, some questions first have to be answered for each tool. As packet creation may cause different loads on the processor depending on which tool is being used, the maximum amount of packets sent by each tool has to be tested. When this is tested, the flooding capabilities of each tool is known and it is possible to set a common maximum value for sending packets through the network. The timing intervals should preferably go in to the microsecond range to be able to measure convergence times with high precision. Secondly, the reliability of a packet generator tool must be tested. This means, different packet transmission intervals must be executed and compared. Preferably, the tests should start at a low transmission interval and end at a higher interval. Since detecting a network convergence in time may require high packet rates, a high transmission interval is preferable. Sending in high intervals however increases the risk of congestion inside the network and will also increase the CPU load significantly. Testing the jitter of the packet creation tools can only be done outside of the network, since packets inside the network is out of testing control. By measuring the gaps between each packet received it is possible to measure both jitter caused by both kernel interrupts and jitter in the physical medium.

The tests which will be executed against the network part is concluded as following:

- Test the amount of packets possible to send
- Test the reliability of packet transmission rate from 25 packets per second to 40000 packets per second or as high as possible.
- Test the jitter interval of an IP packet

The network testing tools will be executed on both general purpose operating systems as well as bare-metal embedded units to compare the performance.

5.6 Bare metal programming

One method of testing creating a traffic generator/analyzer solution for testing is to program micro controlling units without an operating system. This leads to both pros and cons. The main advantage with programming bare metal, is that only the essential functions are implemented, meaning the overhead of the program will be minimal and there is no possibility of getting higher priority interrupts from on-chip peripherals. However, since there is no operating system, which means it will not be possible to schedule tasks with different priorities. This means, scheduling the different parts of the main program on the chip has to be done very carefully so that the program still can be useful. Polling to detect changes in the system has to be done at extreme caution as it will lock the entire system.

5.6.1 Programming a Raspberry Pi

When using an operating system which is not yet fully supporting a hardware platform (in this case ARMv7), the drivers for different on chip peripherals may not be implemented as a library functions in the operating systems API. This leads to bare metal programming, which involves programming the peripheral drivers from scratch using their base addresses to communicate with them. As mentioned before, to test the convergence time in a network, it must be possible to send RAW IP packets, i.e. layers 1, 2 and 3 must be used. The physical layer uses an on-chip USB to Ethernet solution, which converts USB messages to Ethernet messages. Programming bare metal to those kind of solutions requires quite extensive work since both a full USB stack and a full Ethernet stack has to be implemented. Luckily, a feasible open-source solution exist called USPI, which provides drivers for all USB ports, including drivers for the USB/Ethernet port. It also includes a full implementation of a MAC driver, which fetches the MAC address from the Ethernet port of the chip.

Furthermore, an ARP stack is already implemented in the USPI library, which provides the link between layer 2 and layer 3.
5.6.2 Arduino based solution

Another bare-metal programming solution can be used is an Arduino. The Arduino does not support Ethernet by default, but can instead be connected to an Ethernet shield. The Arduino communicates with Ethernet shield using an SPI communication scheme in order to send data using RAW IP sockets. This solution also offers an option to use multicast, which can be useful. Although Arduino only uses a 16Mhz clock, it should still be possible to get an acceptable precision of packet streams. Arduino also offers a large community along with good support for various libraries and functions.

5.6.3 EVK1100 based solution

The EVK1100 board is another bare-metal programming solution, which is slightly more complicated than Arduino. The EVK1100 evaluation board is mainly an evaluation kit which contains an extensive amount of on-chip peripherals. The EVK1100 has an AT32UC3A512 processor with the option of using a 12Mhz clock. The datasheet is very extensive as the processor supports a large amount of General purpose IO functions. The board is officially supported by FreeRTOS meaning a plug and play based solution using atmel studio is possible.
6 Traffic engine development

As previously discussed, many different solutions can be developed to create a complete traffic engine suite. This chapter discusses methods for traffic generation.

6.1 FreeRTOS network stack

Since FreeRTOS already provides a networking library which supports both the TCP and UDP protocols, no new drivers have to be implemented for the EVK1100 and Raspberry Pi 2. The FreeRTOS API also implements the 5 layer based model Internet protocol suite instead of the 7 layer based OSI model. The main task which should be implemented in to the FreeRTOS API is to make the sending and receiving a packet independent of the transport layer (layer4) which means removing the transport protocol functionality and replacing it with a RAW socket functionality. The current structure of the FreeRTOS API is listed below:

1. NetworkInterface.c - This file provides the necessary drivers to the Ethernet port which is attached to the NIC.
2. FreeRTOS_ARP.c - This file implements the link layer and sets a MAC address to a packet.
3. FreeRTOS_IP.c - This file implements the network layer and assigns an IP address to a packet. Here is where the library should be cut off. The overlying layers are not needed since the intention is to create a RAW socket.
4. FreeRTOS_UDP.c - This file implements the transportation layer and sends packets according to the user datagram protocol. An implementation of TCP also exists.
5. FreeRTOS_DHCP.c and more application layer protocols

The prerequisite to implementing this network interface is that drivers for the network interface card are currently existing. These drivers can be fetched from the USPI library previously mentioned. However, the USPI library also implements drivers for the timer inside the raspberry. These interrupt drivers will interfere with the drivers which may lead to very unpredictable results. Furthermore, the USPI library also implements VFP register arguments while FreeRTOS does not. Enabling the VFP (Virtual Floating Point) register arguments in the arm-gnu compiler makes the code run different registers with different properties. This means that it is not possible to combine FreeRTOS and USPI to a feasible solution within a reasonable amount of time, thus making a traffic engine using FreeRTOS on Raspberry Pi an infeasible solution due to time restrictions.

6.2 Bare metal Raspberry Pi

A first test of a traffic engine for Raspberry Pi can be implemented quite simply as the drivers for Ethernet is implemented in the USPI library. The UART driver is however not implemented which leads to an unconventional start procedure whereas the user has to send a packet through Ethernet in order to start the sending sequence. The program flow looks as following:

1. The user inserts all timing parameters directly into the source code of the Raspberry Pi Ethernet file
2. The user compiles the USPI library and copies both the kernel7.img and kernel.img files to the boot directory of the Raspberry Pi SD card
3. The user inserts the SD card in to the Raspberry Pi and powers the Raspberry Pi
4. The user pings the Raspberry Pi in order to start the traffic generation sequence
5. The traffic can be analyzed by using a traffic analyzing tool

The Raspberry Pi bare-metal supports the basic delay functionality which is necessary for generating packets in a fixed sequence. In order to start the sequence, the user must ARPping the Raspberry Pi.
6.3 Arduino UNO with Ethernet shield W5100

The W5100 shield library provides an extensive library on how to send packets through the transport layer. With modifications, the shield can be modified to send various types of IP sockets through a network. The communication architecture of an Arduino based solution is depicted in figure 6.

![Arduino communication architecture](image)

This architecture describes a full Arduino architecture whereas two Arduinos are used, one as sender node and one as receiving node. The upsides with using this solution is that it is a very flexible solution where the data traffic can easily be controlled by a controlling PC. The packet size and stream is also possible to be altered from a controlling PC. Furthermore, the Arduino clock is not based upon an online clock. The timer is instead set according to the point in time where the Arduino first was powered on. Many different solutions can be thought of for this problem, however the most straightforward way would be to use an external real-time clock which connects to both of the Arduinos using an I2C connection.

This solution offers a better usability than the Raspberry Pi solution. The user can easily control the setup of a test through UART which means no code has to be recompiled. However, the measurement of microseconds inside the Arduino will go overflow after 70 minutes and reset to 0.

6.4 EVK1100

The implementation of the traffic generator in EVK1100 was attempted with FreeRTOS and the LwIP library. LwIP provides functionalities for TCP to send and receive messages for embedded systems. LwIP does not fully support raw IP packets, as the initialisation function for their raw API was lacking. Attempts were made to override the current TCP stack of LwIP in order to implement RAW IP sockets. The attempts however failed due to routing failures. Packets that are sent from RAW IP sockets were not able to find a suitable route through the network.

6.5 Packet design

An IP packets are designed to provide necessary information for the packet analyzers. Firstly there are 4 bytes marking the version of the IP packet. IHL marks the header length in bytes.
The fields marked as user defined in Figure 7 is to be set by the user. The TOS field is set to 0 by default, and the total length on the IP packet can be no less than 20 Bytes (header size) + 64 Bytes (Packet sequence no and Packet timestamp). The device marked fields marks properties which depend upon which device is being used.

<table>
<thead>
<tr>
<th>Version</th>
<th>IHL</th>
<th>Type of service (TOS)</th>
<th>Total length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>0x45</td>
<td>User defined</td>
<td>Header length + user defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identification</td>
<td></td>
</tr>
<tr>
<td>0x0000</td>
<td></td>
<td>IP flags</td>
<td>Device</td>
</tr>
<tr>
<td>Time to live (TTL)</td>
<td>Protocol</td>
<td>Header checksum</td>
<td>Calculated</td>
</tr>
<tr>
<td>0x40</td>
<td>0xFF (IP)</td>
<td>Device dependent</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: RAW IP header

The payload information of the packet includes a sequence counter, which is used for calculating packet losses. Without a sequence counter it is not possible to determine if a packet is delayed or lost. There is also a time stamp in the payload, this is required in order to calculate the latency.

6.6 Analyzing a packet stream

Three different terms are used when analyzing a packet stream, listed below:

- Packet jitter
- Packet gaps
- Packet losses

When jitter is mentioned, it is calculated according to Equation 1, Equation 2 and Equation 3. An explanatory picture is depicted in Figure 8. As can be seen, equation 1 will find the highest jitter and Equation 2 will find the lowest jitter. These two equations are used to find the largest jitter for each individual packet. In order to find the overall jitter throughout the entire reading, Equation 3 is used. The average jitter is calculated according to Equation 4.

\[
MaxJitter = \text{Max} (ActualTime - ExpectedTime) \quad (1)
\]

\[
MinJitter = \text{Min} (ActualTime - ExpectedTime) \quad (2)
\]

\[
OverallJitter = MaxJitter - MinJitter \quad (3)
\]

\[
AverageJitter = \frac{\sum_{i=0}^{n} Jitter_i}{n} \quad (4)
\]
Packet gaps are the time from when one packet arrives until the next packet arrives. These are not perfect for finding potential packet losses, however they can suggest when a packet loss might have happened.

The packet losses are found in two ways. Firstly, the analyzer program will look through the data found from the packets sent. This data includes a sequence number which allows the analyzer to see if a packet did not arrive. Secondly, the analyzer will take the time of each packet arriving. If a larger gap happens between two packets at a certain point, it will save this data. The data can then be compared with the packet losses found through the sequence number.
7 Comparison study

This section contains all the comparisons made throughout the thesis. The section contains information on the comparisons made on the real-time operating systems. The packet generator and analyzer tools are compared. These comparisons include results based on time, such as the time between packets received or jitter. Networking protocols are tested towards the end of the section.

7.1 Real-time operating systems

This section contains a comparison of several elements between the RTOSs that are being investigated. These include FreeRTOS, BeRTOS, ChibiOS, Contiki and VxWorks.

7.1.1 Schedulers

The FreeRTOS [11] scheduler supports preemptive, cooperative and round robin scheduling, and it has fix priorities. Round robin is used to distribute a tasks load when there are multiple tasks at the highest priority. The idle task is at the lowest possible priority and it will free the memory of tasks that have been deleted. FreeRTOS also has the possibility to use the POSIX scheduler, making it possible to schedule pthreads[12].

BeRTOS [3] has a preemptive scheduler. If tasks are at the same priority, a round robin policy is used to share the CPU usage. It is recommended for the user to stay within -10 and 10 when choosing priorities, where the larger number is higher priority. This limit is recommended in order to avoid disturbing the background activities such as input processing. The idle task is at the priority of the minimum signed integer value available.

The ChibiOS [6] scheduler is preemptive and allows for up to 128 different priority levels. Several tasks can have the same priority and if there are multiple highest priority tasks then round robin is used to determine which task is running. It is possible for the user to set how long a task will run before the scheduler attempts to put another equal priority task as the running task.

Contiki [8] has a scheduler with less focus on the real-time aspects. While the scheduler allows for real-time tasks and preemption between tasks and interrupts, it also contains processes, which can be run in the background outside of the preemptive mode. The processes do not have any priority and will always be preempted by interrupts or tasks.

VxWorks [1] has two different schedulers, its own default one and the POSIX scheduler. Both of the two schedulers can be configured to always have high priority tasks preempt low priority tasks, but they can also have round robin. Both tasks and pthreads can be scheduled by these schedulers. Tasks and pthreads can be created in both kernel and user space. The main difference between the POSIX and the normal VxWorks scheduling is that only POSIX allows scheduling of pthreads created in a process. Both tasks and pthreads are scheduled according to the chosen scheduling algorithms in all cases except for when pthreads are created in a process and scheduled by POSIX. In this special case, the pthreads can be scheduled according to POSIX FIFO, round robin, sporadic scheduling and the normal preemption priority based scheduling. In any other case, the scheduling will be run according to its priority with either round robin or preemptive scheduling.

From this study it can be concluded that FreeRTOS and VxWorks appear to have the superior scheduler. This is because all of the schedulers have similar functionalities in general, but only FreeRTOS and VxWorks have POSIX support, which puts them ahead.

7.1.2 Memory management

FreeRTOS[13] allows for both static and dynamic memory allocation. As some embedded systems do not support the standard C malloc and free functions, FreeRTOS has its own memory allocation functions in the portable layer. With five different memory allocation algorithms, the user can customize how they wish it to be done. The most basic implementation for them to handle the memory does not allow memory to be freed once it has been allocated, i.e. making it static. The more advanced algorithms wrap the basic C free and malloc functions to make them thread safe, and has ways to avoid fragmentation.
BeRTOS [30] can handle both static and dynamic memory allocation. The RTOS also has support for FAT and BattFS file systems, where FAT is a well-known file system and BattFS is more directed towards performance on embedded systems. There are also functions built into the OS that allocate and free memory [3].

ChibiOS [29] has a core memory manager and a heap memory manager. Both of these have their own functions for memory allocation. The main difference between them is that the core memory manager does not allow freeing of memory, i.e., it makes the memory static. The heap allows freeing of the memory, and the memory allocation for the heap is thread safe. The memory allocator in the heap uses a first-fit strategy.

Contiki [10] supports dynamic memory management and dynamic linking between the programs running on the OS. A managed memory allocator is used in order to avoid fragmentation in the memory, this is done by pushing the memory together. There is also support for memory block management.

VxWorks [27] uses dynamic memory management and in VxWorks 6 they use the best fit algorithm to decide where to allocate memory. No virtual memory is used and there is no paging or swapping, although virtual memory is available as an add-on.

To conclude this section, it can be seen that FreeRTOS, BeRTOS and ChibiOS have static and dynamic memory management, while Contiki and VxWorks are dynamic only. As both static memory management and dynamics is preferred, FreeRTOS and BeRTOS are preferred for this.

7.1.3 Preemption protocols

Different OSes often offer a solution for disallowing priority inversion to happen during mutual exclusion of a variable. These solutions are typically semaphores with some extended functionality such as priority inheritance protocol (PIP), priority ceiling protocol (PCP) or immediate priority ceiling protocol (IPCP).

<table>
<thead>
<tr>
<th>RTOS</th>
<th>PIP</th>
<th>PCP</th>
<th>IPCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeRTOS</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BeRTOS</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VxWorks</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ChibiOS_RT</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Contiki</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6: anti priority inversion protocols

As seen in the table above, all RTOSs but Contiki provide support for the priority inheritance protocol. However, support for PCP or IPCP would have been favorable due to better performance.

7.1.4 Documentation

The documentation is a very important part regarding software development in general. A good documentation will decrease the development process of a program greatly, since it will be much easier to find necessary information. Below follows a short survey of the documentation which can be found for each RTOS.

FreeRTOS provides a very extensive documentation from their website. There are guides for installation, API documentation and benchmark tests. FreeRTOS also has a forum where the community can share related code.

BeRTOS provides a documentation through Github. This means it will be difficult to find certain information with web based search engines, as the documentation is not found online. Their documentation provides a user with information on how to get started with the installation. It also contains a large API documentation which lists functions and data structures.

ChibiOS’s documentation does not contain a getting started guide, unlike the previous mentioned operating systems. The documentation has an extensive API with a long list of the data structures available in the RTOS, as well as the class hierarchy. The main page contains a short summary of what ChibiOS has to offer.
Contiki has a few example codes that could make it easier to get started, however it does not have a guide for the installation or recommended steps for a beginner. There is an extensive list of the data structures and functions provided by the RTOS.

VxWorks documentation has detailed descriptions on how to use the functions that exist in the real-time operating system. There are guides on how to install the real-time operating system on different file systems. The documentation includes some small samples of example code.

Since it is the documentation that has been compared in this section, it should be mentioned that the opinion of which RTOS is superior is subjective. However, the more solid points of what documentation contains the most information could be somewhat measured. This means FreeRTOS is the superior choice, as it has more features than ChibiOS, Contiki and BeRTOS lacks the internet availability that FreeRTOS has.

7.2 Open source packet analyzer tools

In the following sections, a traffic generator and a traffic analyzer will be selected. This applies to both user space based tools and kernel based tools.

7.2.1 Kernel based tools

Due to the lack of kernel based tools, the comparison over KUTE and Pktgen-txrx will be brief. KUTE is a kernel based tool which aims to be a maximum performance traffic generator and receiver for gigabit internet. KUTE uses UDP packets to generate and receive packet streams.

The second tool under kernel based tools is called Pktgen originally was developed by Intel under the name dpdk-pktgen and could only be used as a traffic generator. It is also included as standard in the Linux kernel. In 2010, Torrents [32] created an updated version of Pktgen which also contained a receiving end of the program. Pktgen-txrx focuses on creating and sending packets from multiple cores which makes it possible to send a very high network load.

7.2.2 User space tools

hping3 [17] is a tool which is made primarily to analyze packets. TCP, UDP, ICMP and RAW-IP are supported, and in the past it was used as a security tool. However it can now also be used for firewall and network testing, and several other things. hping3 runs without problems on the 3.19 Linux kernel, and it runs on Linux, FreeBSD, NetBSD, OpenBSD, Solaris, MacOs X and Windows.

Mausezahn [14] is another tool that can be used when analyzing a network. Like hping3 it is able to generate packets and in this case it is the tools main purpose to act as a traffic generator. On a performance test performed at least 5 years ago, they reached a 755 Mbit/s with the packages that sent from Mausezahn. Currently Mausezahn is included in netsniff-ng’s tool package, and they maintain it for newer releases. It has been tested and worked on the unix kernel version 3.19.

TShark [38] is a network protocol analyzer that uses many of the same modules that are used by the more well known program Wireshark. The main difference between the two programs is that TShark is entirely run in the command line interface, making it more suitable for this thesis work. TShark can read data from a network while there is traffic going on in it and it can also open old files to analyze old saved data. The default format for saving with TShark is pcap.

Ngrep [26] is a tool that can be used for analyzing networks which is has a similar functionality to the standard GNU grep function, but for networks. Currently it supports IPv4/6, TCP, UDP, ICMPv4/6, IGMP and Raw across Ethernet, PPP, SLIP, FDDI, Token Ring and null interfaces. Ngrep is also able to use pcap for saving the data that is being read.

7.3 Evaluation summary

Based on the parameters investigated in this section, suitable traffic-engines and real-time operating systems have been chosen. A summary of the properties which are valued is listed in Table 7.

As seen in the table, FreeRTOS provides all functionality that may be needed for the testing system. A preemptive scheduler is required for real-time capabilities, as a non preemptive scheduler will not be able to adapt to rate monotonic scheduling protocol. Furthermore, a POSIX scheduler is also a preferable option since it provides the pthreads library. Using the pthreads library will
Table 7: anti priority inversion protocols

<table>
<thead>
<tr>
<th>RTOS</th>
<th>Preemptive</th>
<th>POSIX</th>
<th>Static</th>
<th>Dynamic</th>
<th>PIP</th>
<th>Doc</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeRTOS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Best</td>
</tr>
<tr>
<td>BeRTOS</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Third best</td>
</tr>
<tr>
<td>VxWorks</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Not available</td>
</tr>
<tr>
<td>ChibiOS_RT</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Second best</td>
</tr>
<tr>
<td>Contiki</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Fourth best</td>
</tr>
</tbody>
</table>

Making the system easier to port to another POSIX compatible platform such as Linux. For memory management, static memory allocation is needed since the payload of the packets is going to be static. Changing the payload during runtime could be a nice feature, it is not as crucial. Having an anti priority inversion protocol is required for better real-time capabilities. Lastly, a good documentation is very preferable since it allows for fast learning thus decreasing the development time needed.

Table 8 shows the capabilities of the kernel based tools. Pktgen-txrx supports newer kernel versions than KUTE which makes it a more modern solution. The regular Pktgen is also an included tool inside the regular Linux kernel, and since Pktgen-txrx inherits its function from this it could be considered a great benefit.

<table>
<thead>
<tr>
<th>Traffic engine</th>
<th>Kernel support</th>
</tr>
</thead>
<tbody>
<tr>
<td>KUTE</td>
<td>2.6</td>
</tr>
<tr>
<td>Pktgen-txrx</td>
<td>2.6.36, 3.11, 3.18, 3.19</td>
</tr>
</tbody>
</table>

Table 8: Kernel based tools kernel support

7.4 Optimizing user space tools

Although the user space tools are being run within a non-predictable environment a few things can be done in order to accomplish more predictability. Standard Linux offers the opportunity of setting priorities as start flags to a running process using the nice(prio) flag. By running a program with this command, it is at least possible to guarantee that the traffic engine will obtain the highest priority of all user space applications run. Linux also offers another opportunity to run programs in a more real-time way. Using this real-time kernel instead of the regular round robin kernel, it may be possible to obtain more real-time capabilities.

7.5 Traffic engine test cases

Different test cases for the traffic engine which were executed come as following:

- Performance test 1: Flooding, testing the sending frequency;
- Performance test 2: Testing TOS priority of a packet;
- Reliability test 1: Execute test sending packets with the interval of 1 millisecond;
- Reliability test 2: Test packet generation rates 25; 50; 100; 125; 250; 500; 1000; 2000; 4000; 8000; 10000; 20000; 40000;
- Reliability test 3: Number of packets that are sent and received;
- Usability evaluation

The traffic tests cases are executed on a network with no load.
7.6 Performance test 1

The flooding test of a network is used as an initial test of the traffic engine. By executing this test, it is possible to see the actual performance of the traffic generators. As detecting a network convergence may require a certain amount of packets per second to be detected, this test is crucial for deciding which approach is suitable. The test procedure is illustrated in Figure 9.

The results come as presented in Table 9.

![Figure 9: Performance test 1 procedure](image)

<table>
<thead>
<tr>
<th>Packet generator</th>
<th>Average maximum pps</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry Pi (Bare metal)</td>
<td>44821</td>
<td>Raspberry Pi</td>
</tr>
<tr>
<td>Mausezahn RT preempt</td>
<td>16873</td>
<td>Raspberry Pi</td>
</tr>
<tr>
<td>Mausezahn</td>
<td>23683</td>
<td>Raspberry Pi</td>
</tr>
<tr>
<td>Arduino</td>
<td>2236</td>
<td>Arduino</td>
</tr>
<tr>
<td>Pktgen-txrx</td>
<td>N/A</td>
<td>Raspberry Pi</td>
</tr>
</tbody>
</table>

Table 9: Tests with the traffic generators, picked up with Wireshark. Raspberry Pi 2B is used for every solution except for the Arduino solution, where Arduino Uno was used instead.

7.7 Performance test 2

Setting TOS priorities is an important property of an IP packet as it decides the actual priority of a packet through the network. If it is possible of setting the TOS priority of a packet in a packet generator, the network under test must be able to handle this, else something is fundamentally wrong with the network IP protocol implementation. TOS compatibility is listed in table 10.

7.8 Reliability test 1

This test is to determine the reliability of a traffic generator. One packet should be sent every millisecond and received at the same rate. This test is done to test the real-time capabilities of the system. The test procedure can be seen in Figure 10.
<table>
<thead>
<tr>
<th>Packet generator</th>
<th>TOS configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry Pi (Bare metal)</td>
<td>No</td>
</tr>
<tr>
<td>Mausezahn RT preempt</td>
<td>Yes</td>
</tr>
<tr>
<td>Mausezahn</td>
<td>Yes</td>
</tr>
<tr>
<td>Arduino</td>
<td>Yes</td>
</tr>
<tr>
<td>Pktgen-txrx</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 10: Tests with the traffic generators, picked up with Wireshark

Mausezahn and Pktgen-txrx have been tested against the network analyzer tool TShark. This was performed from two different Raspberry Pi 2B towards another Raspberry Pi 2B hosting the TShark program. One Raspberry Pi 2B used Ubuntu and hosted Pktgen as well as Mausezahn. The other Raspberry Pi 2B ran the operating system Raspbian Jessie with the Linux RT preemption patch, and Mausezahn could be tested on this. Pktgen could not be run on the Raspbian Jessie as it is only supported on few kernels. One packet generator has been developed on a Raspberry Pi 2B without an operating system and another packet generator has been developed on an Arduino UNO.

Table 11 shows the results of the previously mentioned tests. Mausezahn, with and without the preemption patch, could never accurately hold 1000 packets per second. However, the relatively low jitter and biggest gap values in the Mausezahn tests suggests that even while it cannot delay the messages the correct amount, it seems to be stable. Pktgen, Arduino and the bare metal solution on the Raspberry Pi 2B appear to be close to equal in their accuracy at reaching 1000 messages per second, however Pktgen seems to often have the packets get delayed more than they should, suggesting some kind of interference.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Biggest gap</th>
<th>Biggest Jitter</th>
<th>Packets per second</th>
<th>Average Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mausezahn</td>
<td>0.004878</td>
<td>0.003878</td>
<td>891.792</td>
<td>0.000124</td>
</tr>
<tr>
<td>Mausezahn with preemption patch</td>
<td>0.004824</td>
<td>0.003824</td>
<td>888.915</td>
<td>0.000125</td>
</tr>
<tr>
<td>Pktgen-txrx</td>
<td>0.006191</td>
<td>0.005191</td>
<td>1000.015</td>
<td>0.000001</td>
</tr>
<tr>
<td>RPI Bare Metal</td>
<td>0.001113</td>
<td>0.000202</td>
<td>1000.026</td>
<td>0.000003</td>
</tr>
<tr>
<td>Arduino</td>
<td>0.001118</td>
<td>0.000916</td>
<td>1001.058</td>
<td>0.000005</td>
</tr>
<tr>
<td>Ixia</td>
<td>0.001187</td>
<td>0.000365</td>
<td>1000.005</td>
<td>0.000004</td>
</tr>
</tbody>
</table>

Table 11: Tests with Pktgen-txrx, Mausezahn with and without preemption patch, Raspberry Pi bare metal and Arduino picked up with TShark. The biggest gap, biggest jitter and average jitter are measured in seconds.

To get a comparison of TShark as packet receiver, the Ixia hardware was used for receiving packets. The results of average packet jitter is shown in Table 12.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Biggest gap</th>
<th>Biggest Jitter</th>
<th>Packets per second</th>
<th>Average Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mausezahn</td>
<td>0.004743</td>
<td>0.004676</td>
<td>894.904</td>
<td>0.000118</td>
</tr>
<tr>
<td>Pktgen-txrx</td>
<td>0.007320</td>
<td>0.007297</td>
<td>1000.025</td>
<td>0.000004</td>
</tr>
<tr>
<td>RPI Bare Metal</td>
<td>0.001014</td>
<td>0.000028</td>
<td>1000.021</td>
<td>0.000001</td>
</tr>
<tr>
<td>Arduino</td>
<td>0.001009</td>
<td>0.000763</td>
<td>999.952</td>
<td>0.000001</td>
</tr>
<tr>
<td>Ixia</td>
<td>0.001000</td>
<td>0.000002</td>
<td>1000.012</td>
<td>0.000001</td>
</tr>
</tbody>
</table>

Table 12: Tests with Pktgen-txrx, Mausezahn with and without preemption patch, Raspberry Pi bare metal and Arduino picked up with Ixia. The biggest gap, biggest jitter and average jitter are measured in seconds.

### 7.9 Reliability test 2

Sending packets each millisecond is a good benchmark for packet generators. However, as mentioned before, faster packet streams may be needed in order to detect a network convergence. This test shows the reliability of a packet stream by generating a stream which will increase to a microsecond range. The tests will produce two type of graphs, one which will present the packet jitter, i.e. the time difference between when the packet should have been sent until it actually was sent. The other graph will present the packet deviation, which gives how many packets the stream deviates from the desired stream. These graphs show the jitter and packet deviation at 2000 packets per second and below. This limitation is because the Arduino hardware cannot go much further than that amount of packets per second. A graph including the tests up to 40000 packets per second would make it very difficult to see any meaningful results in the graph comparisons.

As can be seen in the Arduino tests presented in Figure 13 and 12, the Arduino based solution gives very accurate packet streams. The maximum jitter measured was slightly below 0.04 milliseconds at a packet stream of 25PPS. This is a very good result and proves that Arduino very much can be a feasible solution regarding reliability. Furthermore, the packet deviation of the Arduino was also very minimal, this could however probably be calibrated in order to gain an optimal level.

The reliability tests performed with Mausezahn with preemption patch are presented in Figure 13 and Figure 14. The Mausezahn with preemption patch packet generator can reach a total of 16873 packets per second, as seen in Table 9 when it has no delay between the messages that are sent, however it never reached above 10000 packets per second when told to reach above this limit. Figure 14 suggests a deviation that increases more as the packets per second are increasing.

Figure 13 and Figure 15 present the data from the Mausezahn tests done without the preemption patch. Figure 13 shows a curve that increases the packet deviation as the rate at which the packets are sent get increased, even while staying below the maximum rate that the Raspberry Pi 2B could send packets at with Mausezahn. Figure 15 shows the maximum jitter found in the tests with Mausezahn, showing an unstable decrease from 0.04 seconds of jitter at the slower rate tests.
towards the 0.005 seconds of jitter when the program was sending at 2000 packets per second. Not only does Mausezahn fall far behind when attempting higher packet rates, it also appears to have a very large and unstable amount of jitter.

The tests made with Pktgen are presented in the graphs in Figure 13 and Figure 16. The jitter presented in Figure 13 peaks at slightly above 0.005 seconds and the second highest reading at slightly above 0.003 seconds, the Pktgen solution can be viewed as reliable. Figure 16 suggests similar results, as the intended amount of packets per second never deviates far from the actual packets per second. It should however be noted that Pktgen failed to send at the slowest rate tested, 25 packets per second, and it also failed at sending at above the rate of 2000 packets per second.

The USPI based solution seen in Figure 13 and 17 were the only generators which could generate packets at a high speed with reasonable reliability. The maximum packets which could be generated by using a delay functionality was 27000 packets per second before going unstable, which is more than enough for detecting a network convergence. To make the delay functionality easy, a fake clock scaled 1MHz was used, which means the precision of the clock was down to one micro second per tick. This could be the reason why it was not possible to achieve the full 44821 PPS which was tested in the maximum PPS test. With a maximum jitter of 0.003 seconds, the USPI can be deemed as a very reliable solution. The packet difference provided by USPI was also very low, with a maximum of 0.19 packets. However, since the graph is increasing steadily, it can be assumed that the packet deviation would increase if the packet stream were able to sending more packets per second.

Finally, the average jitter produced by each generator is presented in Table 13.
Figure 12: Arduino packet deviation

<table>
<thead>
<tr>
<th>PPS</th>
<th>Arduino</th>
<th>MZ</th>
<th>PKTgen</th>
<th>USPI</th>
<th>MZP</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.001625</td>
<td>0.01839</td>
<td>N/A</td>
<td>0.000003</td>
<td>0.000134</td>
</tr>
<tr>
<td>50</td>
<td>0.000417</td>
<td>0.000545</td>
<td>0.000004</td>
<td>0.000003</td>
<td>0.000132</td>
</tr>
<tr>
<td>100</td>
<td>0.000107</td>
<td>0.000238</td>
<td>0.000006</td>
<td>0.000003</td>
<td>0.000132</td>
</tr>
<tr>
<td>125</td>
<td>0.003015</td>
<td>0.003138</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003128</td>
</tr>
<tr>
<td>250</td>
<td>0.001503</td>
<td>0.00163</td>
<td>0.0015</td>
<td>0.0015</td>
<td>0.001627</td>
</tr>
<tr>
<td>500</td>
<td>0.000754</td>
<td>0.000877</td>
<td>0.00075</td>
<td>0.00075</td>
<td>0.000877</td>
</tr>
<tr>
<td>1000</td>
<td>0.000005</td>
<td>0.000124</td>
<td>0.00001</td>
<td>0.000003</td>
<td>0.000125</td>
</tr>
<tr>
<td>2000</td>
<td>0.000006</td>
<td>0.000123</td>
<td>0.000007</td>
<td>0.000002</td>
<td>0.000123</td>
</tr>
</tbody>
</table>

Table 13: The average jitter (seconds) found from several different packet per second measurements

7.10 Reliability test 3

This test was executed in order to test the reliability of the network switches. There is always a possibility that packets get lost in a network, especially when there is no transmission control protocol. This test is executed to see if there is any difference in packet losses when using different types of packet generators. The packet streams were set up to use a delay of 1 millisecond (i.e 1000 packets per second) and generate 60000 packets. The results of this test can be seen in Table 14

<table>
<thead>
<tr>
<th>Packet generator</th>
<th>Average packets lost</th>
</tr>
</thead>
<tbody>
<tr>
<td>USPI</td>
<td>0</td>
</tr>
<tr>
<td>Mausezahn RT preempt</td>
<td>0</td>
</tr>
<tr>
<td>Arduino</td>
<td>0</td>
</tr>
<tr>
<td>Mausezahn</td>
<td>0</td>
</tr>
<tr>
<td>PKTgen</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 14: Tests with pktgen and mausezahn, picked up with Wireshark

7.11 Usability evaluation

The USPI solution provided very good performance results, but is much less user and developer friendly. The documentation is very poor and does not use the common C libraries, this mean
standard functions of the C library has to be implemented by the developer. Furthermore, the
compiling and running a program is very time consuming since the SD card has to be moved
between the computer and the Raspberry Pi. This leaves little room for writing complex code, as
the debugging process simply would take too long time.

Arduino on the other hand provide a very well documented library for writing both simple
and more advance programs. Since an Arduino sketch can be uploaded to an Arduino through
USB, no extra equipment is needed and makes the development pace very fast. Serial port is
already implanted to be compatible with the USB port, which makes sending serial messages to
the Arduino serial reading program a very easy task. The only thing which Arduino lacks regarding
development usability is a live debugger.

Even though it was not possible to implement a traffic generator for EVK1100, it is still
important to mention the usability if other attempts would be made. In order to program this
device an external device called AVR dragon has to be used. This debugger enables a stepping
functionality when debugging a program. This makes the EVK1100 a very convenient solution if it
would have worked. As mentioned before, FreeRTOS could also be uploaded straight to the board
without any porting needed.

Mausezahn is very simple to get running, requiring only a command in the terminal and an
internet connection in order to install it. The program has a help command, showing all the
available commands, however it does not give a deeper explanation. There does however seem to
be a large amount of people using the program, this makes it very easy to find solutions to problems
and examples on how to run the program. Mausezahn is run by writing commands following the
"mz" keyword in the Linux terminal, where things such as the IP to send to or the delay between
each package can be specified.

The usage of Mausezahn RT preempt is the same as regular Mausezahn with the exception of
the RT Linux kernel which has to be used. Finding information on how to compile the Linux kernel
was difficult, as a compatible RT preempt kernel has to be added on to an existing Linux branch.
Furthermore, the configuration files of the kernel have to be changed, which however was not a
difficult process. The kernel however only has to be changed once which makes little impact on
the development time of a test. Using the RT kernel will change the kernel name of the operating
system thus making Pktgen uncompatible with the system.

Pktgen is used through a bash script that sends messages to the kernel in how it should be used.

![Figure 13: Jitter comparison. More detailed graphs can be found in Section A.5](image)

These scripts allow a customization on how large the delay should be between each message, how many cores should be used, what MAC and IP address should be sent to and similar specifications. Each time the device is rebooted, one of the files must be inserted into the kernel and a script must be run in order to configure the Ethernet settings.

7.12 Network performance tests

As seen in the tests in Section 7.5, the results between the different traffic generators and analyzers varies. The most suitable approach may vary depending on which switches are being used. The switches which are being used in this study is as previously mentioned the Westermo RedFox switch. The switches are connected in to a ring of four switches with one link serving as backup link. To test the convergence time, an Ethernet link breaker was attached to the main middle link, as seen in Figure 18.

Different tests can be applied in order to measure the performance of a network during a convergence. In this thesis we measure the Jitter and packets lost through a network.

Executing a test towards layer 3 protocols can be done using a similar setup. The RedFox devices have to be configured as routers instead of switches.

7.13 Measuring network convergence

Three different techniques can be used for finding the network convergence time. When a network link goes down, three different factors will be affected:

- Packet losses - Packet losses may occur since the packet buffers may become full.
- Packet gaps - The gaps in which packets are received will increase when the network converges
- Jitter - The jitter of the packets will increase

To force the a communication link to fail between the Ethernet cable connecting switch L2 and L4 will be pulled, which will force the system to converge using switch L3 instead of L2. The measurement of a network convergence will be based upon using a jitter threshold, whenever the jitter of a packet stream surpasses this threshold, network is considered to be broken and the convergence time will be calculated according to Equation 5.
The Arduino packet generator solution is used to send 300000 packets to the Raspberry Pi 2B with TShark running, recording all the packets received. The packets are sent at 1000 packets per second. During the testing run for each protocol, an Ethernet cable is pulled out in order to force a rerouting through the network.

A RAW IP packet receiver is built as a normal application running on Linux, on Raspberry Pi 2B. The reason for this RAW IP receiver is to verify that the data interpreted by the Python analyzing program is correct. Currently it has one potential major issue for future tests, which is the processing power. With how each packet is received and then processed, if it receives too many packets in a too short time it will be flooded. In such a case the the buffer will fill before the packets are received, which will lead to packet loss.

7.13.1 Packet losses

It is important to measure the packets lost during a convergence. The smaller amount of packet losses during a convergence, the better. However, the results of the networks varies a lot depending on how fast the networks converge. The layer 2 networks has a hardware buffer which saves queues packets which cannot be sent. The hardware buffers are however not very large, and may lose packets when the network has a high traffic. Layer 3 networks can also lose packets which makes this an important measurement. In order to measure the packets lost, a stream of 1000 packets per second is being sent through the network. A raw IP receiver is being used on the other end to analyze the packet sequence counter. Real-time capabilities is not needed in this scenario, wherefore a Raspberry Pi with an IP raw receiver is used for simplicity. At random times, the Ethernet cable connected to the main route of the network will be disconnected, thus forcing the network to converge.

The packet losses which were detected are presented in Table 15 were the packet loss interval is presented.

7.13.2 Protocol jitter measurement

Even though the jitter was already tested between different tools, it may still vary between different protocols as their forwarding mechanisms are different. Table 16 shows different jitter parameters
for each protocol measured with Arduino as packet generator and TShark as traffic analyzer.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Calculated maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRNT (Layer 2)</td>
<td>0.005022</td>
</tr>
<tr>
<td>RSTP (Layer 2)</td>
<td>0.221051</td>
</tr>
<tr>
<td>RIP (Layer 3)</td>
<td>30.475858</td>
</tr>
<tr>
<td>OSPF (Layer 3)</td>
<td>47.898342</td>
</tr>
</tbody>
</table>

Table 16: Biggest jitter found during network convergence, measured in seconds.

7.13.3 Protocol worst case convergence time

The gaps in time between two received packets are an important factor when measuring the convergence time of a system. A large gap between two packets means that something slowed the system down. If packets were lost at the same time, there is a large chance that a rerouting happened. If a packet loss can be spotted at the larger time gaps it can be concluded that a rerouting happened. This means that the time of the gap is the convergence time.

In order to know how reliable a protocol is, it is important to know what kind of worst case convergence times can be expected. This section presents the worst case convergence time in a series of tests on four different protocols. The convergence time is found by measuring the gaps in time between packets, and if one is found that is larger than current threshold, a convergence will be detected.

Figure 19 and Figure 20 present the worst case found in packet losses from the first measurement. Figure 19 presents the results for the router protocols while Figure 20 presents the results from the switching protocol tests. The routing protocols RIP and OSPF are compared where RIP
is below 20 seconds in its worst case while OSPF is at 44 seconds. From the switching protocols FRNT has a worst case convergence time of 22 milliseconds and RSTP reached 49 milliseconds.
Figure 18: Arduino convergence test architecture

Figure 19: Worst case for the convergence time found in the first test for the routers
Figure 20: Worst case for the convergence time found in the first test for the switches
8 Results & Conclusion

This section describes the results and conclusions from this thesis followed by the future works.

8.1 Summary of results

Table 11 reveals that the Arduino and USPI solutions are the superior tools when it comes to the gap in between the packets sent. Mausezahn with and without preemption patch and Pktgen fell behind in comparison. The jitter found among the tools provided similar results, except that the USPI solution proved to have less jitter than the Arduino solution.

Tests were performed to compare the maximum speed of the packet generators, the results are seen in Table 9. Arduino is by far the slowest, because of its lacking hardware. Mausezahn did close to half the speed of the USPI solution. Pktgen could not join this comparison as any test above the rate of 2000 packets per second made it shut down.

When the jitter between each packet was compared and then presented in Figure 13 it was found that the USPI solution had very low jitter compared to the rest. Otherwise the rest were close to equal in their jitter except for from Arduino and Mausezahn. They had most of their readings similar to the results of the other packet generators, except these had a much bigger jitter at lower values. Table 11 shows that only RPI Bare Metal and Arduino can be compared to the Ixia packet generator. Table 12 shows how the tools performed when recorded by the Ixia packet receiver. RPI Bare metal and Arduino both got an average jitter similar to that of the Ixia. However, it should be mentioned that their average jitter was superior to when they were received on TShark.

In the usability evaluation, it is found that the Arduino solution and, if the EVK1100 solution had worked, both have the advantage of a very simple programming tool. USPI falls far behind in this evaluation as it must compile a new kernel each time the code is changed. Mausezahn ranks high in this evaluation too as there are no difficulties with running the program. Pktgen can be run in a similar fashion as Mausezahn. However before Pktgen can be run some files must first be inserted into the kernel, making the usability less.

In Table 15 the results of the packet loss test on the protocols can be seen. FRNT appears to be far superior to RSTP when it comes to the packets lost during a rerouting in layer 2. In layer 3, RIP provided a much lower packet loss than OSPF. The maximum jitter test results can be found in Table 16. The results show that FRNT is superior to RSTP in the layer 2 comparison. RIP provides a lower maximum jitter than OSPF.

To summarize the results according to the research questions, the most suitable real-time operating system is FreeRTOS. While all the investigated real-time operating systems had what was required, FreeRTOS had extra features such as the posix scheduler, and it has the largest community. The larger community will make it easier to find information, as forums will be full of useful information. The only investigated real-time operating system that could be compared to FreeRTOS is VxWorks. It will however not be considered as it is not open source.

Of the investigated switching protocols, FRNT has a superior convergence time compared to RSTP. As for the routing protocols, RIP has a superior convergence time compared to OSPF. This can be seen in Table 16. The performance is measured in convergence time on a ring connected network.

To measure the convergence time in real-time we are dependent on two crucial factors. Jitter inside the packet generator and jitter inside the packet receiver. From the test it was found that the jitter is very small in most solutions, however not predictable enough for hard real-time systems. For achieving an acceptable amount of jitter we must measure the convergence time by using either bare metal or kernel based packet generators, as can be seen in Table 13 where the average jitter is shown. However, for packet receivers it is enough to sniff the network card in user space, this can be seen in Table 11 and Table 12, where Tshark performs similarly to Ixia.

8.1.1 Automated test system

The purpose of this thesis was to develop a test system feasible of executing network convergence test with an acceptable performance. The system model we chose for the test system is depicted in Figure 21.
In the test system, Fawlty gives the test execution order through a Python script. The parameters include setting packet sending frequency, packet size and amount of packets to be sent. The packet analyzer will also be started from Fawlty at the same time the packet generator started. The packet analyzer will run for a calculated amount of time before stopping. When the test sequence is finished, the analyzer will analyze the data received and send back an analyzed file containing jitter, packet gaps and packets lost. This data allows for calculating the convergence time of a network.

8.2 Conclusion

As proclaimed earlier, FreeRTOS did not work for the Raspberry Pi solution. This causes no effect to the traffic generator part, as it is not dependent on running multiple tasks. The receiving end could however benefit from a real-time scheduler as it could potentially increase the amount of packets which can be received. More tasks could also be added to handle the IO parts, which would mean the user could be monitoring packets in real-time. However, the only possibilities of revealing data to the user on the Raspberry Pi is to either save it on to the SD card or save it in the RAM memory and display it to the user as soon as the test is done. Saving the data in to the RAM memory is obviously not a feasible method, as it at some point could overflow. Saving the data to the SD card without an operating system must be done after a packet is received, thus taking processing time from the packet receiving task. FreeRTOS could have utilized the cores of the Raspberry Pi and parallelized this job which would allow more packets to be received. FreeRTOS would also have provided a more complete network stack which is much easier to use. Since it was not possible to combine FreeRTOS with USPI, the only possibility was to use only one of these solutions. Implementing a network interface for FreeRTOS would have been too time consuming, thus the choice fell onto using a USPI solution.

Using a bare metal Raspberry Pi as traffic generator instead of Arduino could potentially be a very good solution in the future. USPI provides an interface to the hardware, which makes it possible to send packets through the Ethernet port. The Raspberry Pi provides a much better CPU than an Arduino which makes it possible of sending more packets per second, this can be a good property in the future as it will allow for greater precision in measuring the convergence time. The drawbacks with USPI is as mentioned before, limitations with programming and debugging.

When researching kernel-based packet engines, we chose to investigate pktgen with a pktgen receiving patch. This proved to be a mistake, as it did not work as well as expected. The reliability of the engine was very low as it tended to crash the operating system at random times.

The EVK1100 results were very disappointing. Even though TCP packets could be sent from the EVK1100, it could not provide neither a valid RAW IP stack or UDP stack. Even though
the EVK1100 only has a 12 MHz processor compared to Arduino’s 16MHz, no conclusions can be drawn regarding which result the EVK could have accomplished. Since the EVK provide a direct link from the chip to the Ethernet port while the Arduino provide an SPI connection to the Ethernet shield chip, it is not possible to speculate about the outcomes.

Regarding the packet generators maximum sending frequencies, we could see large differences depending on which one was tested. Using a USPI was by far the fastest type of solution, this result may seem a little obvious, as it uses the fastest processing power along with no operating system. What was surprising regarding these test was that Mausezahn was faster than Mausezahn with an RT preempt patch. This may have been because Mausezahn is not high enough priority in the system.

The maximum jitter of the packet generators proved to be very stable. It was very surprising that Mausezahn managed to achieve less maximum jitter than Arduino, since it will get preempted by the kernel. It was expected that the Arduino would generate more jitter than the bare metal solution because of the difference in the processing power. It was expected that the Mausezahn with preemption patch solution should get less jitter than Mausezahn without preemption patch.

From Table 11 it is possible to see that Mausezahn with preemption patch got far more packets at half the intended speed than the Mausezahn without preemption patch did, which was surprising, as the opposite was expected. Perhaps there is something occasionally interfering with the program in the preemption patch. Mausezahn could be said to be a reliable tool that will send packets at the specified time, as long as the rate at which they are sent is not above 100 per second. Everything above 100 per second on any of the Mausezahn solutions provided a bad result, and it kept getting worse the higher the rate got. One possible explanation as to why Mausezahn failed at sending the correct amount of packets per second could be the delay time. Perhaps the developers calculated it wrongly. Only Pktgen got an exceptionally high jitter in comparison to what was expected, and it turned out to be a big disappointment because of how easily the program gave up. It failed at all attempts at going above 2000 packets per second, and often during a run it would crash without warning.

Another jitter measurement done was calculating the average jitter of a packet stream. By measuring the average jitter, it is easier to compare the performance of the different packet generators. In this case, we get a better comparison of what the usual jitter actually is. By executing this study, it is easy to conclude that USPI is the most reliable packet generator, followed by Arduino and Pktgen. What is very interesting in these results is that packet experience a heavy increase in jitter at 125 PPS compared to 100 PPS. Since all packet generator types experience this, it cannot be explained as kernel interrupts. However, due to time restrictions, this issue could not be investigated further.

The jitter through a network was first thought of as a problem, but later on seen as an opportunity. Rather than thinking of jitter as something which may cause calculations go wrong, jitter was used to calculate and detect the convergence of a system. Since the jitter was so small and stable in the packet generators, this was a possible outcome. To verify this conclusion, packet losses and gaps between packets received were also calculated.

The protocol tests proved to be similar to the expectations. The test focusing on finding a worst case convergence time showed that FRNT is quicker than RSTP, less than half of its worst case. This shows that in a switch network of four switches, such as the one used here, FRNT is a superior protocol when the worst case convergence time matters. It should be noted that only one reading from the FRNT tests showed 22 milliseconds, all other readings were 8 milliseconds at worst. Similar results came from RIP and OSPF, the routing protocols, where the convergence time of RIP was less than half of OSPF’s convergence time in the worst case found during this test. This shows that RIP is the superior router protocol when the current 4-router ring setup is used. However, the effects of adding more routers could potentially change these results. The techniques presented by Vidalenc et al. [33] could perhaps change the order of which would be the most efficient. Their technique for improving the convergence time is supposed to predict when an error occurs, this may have made the slower solutions speed up. Basu and Riecke [2] suggested a solution for improving upon OSPF. This has not been used for these tests, however it could prove valuable to see if OSPF would improve with these improvements.

As of these results, the FRNT technology was revealed to be more stable and efficient than the more commonly used RSTP technology. This was obviously a very good result for Westermo,
as it proves that simple methods can achieve great results. This opens up new doors to research, as FRNT possibly could be introduced as a standard in IEEE. However, further research is needed regarding the FRNT protocol, which can prove the usability of FRNT in larger networks.

Seeing that the convergence of RIP was faster than OSPF was not a big surprise as it uses more simple mechanism for finding a suitable route through the network. In networks where re-routing is commonly occurring, RIP obviously is more suitable than OSPF. However, when convergences often happen, something should be clearly wrong in the network, and it should be replaced. The related work revealed that OSPF should find a faster route through a network than RIP, however since latency was not implemented, this could not be proven. A routing network could possibly use the same setup as the switching network - a ring based solution with 4 routers. This would allow the FRNT to be altered and implemented as a routing convergence protocol as well which could improve the convergence time of routers drastically.

The 1 ms second tests include an Ixia comparison, whereas Ixia is used as both packet sender and receiver. The test against an Ixia was made in order to test our proposed traffic engine suites against methods which are commonly used today. When TShark is used as receiver, we generate a small amount of jitter. Ixia generated a jitter of 4 micro seconds, which is very low, however only slightly better than Arduino. When using Ixia as receiver, we see a clear decrease in jitter, where Arduino, USPI and Ixia managed to generate a minimum jitter of 1 micro second. This mean, that TShark may experience some delay due to kernel interrupts, however very small.

According to Westermo’s internal references, Ixia is considered as standard among network measurement tools. It was very pleasing to see that cheap traffic generator solutions such as Arduino and Pktgen can be used to achieve similar results. The packet analyzing performance of TShark was also sufficient compared to the Ixia. This means a whole traffic analyzing suite with acceptable performance can be created at a very small expense.

8.3 Limitations

The main limitation of this project is that the investigated operating systems will have to be a real time operating system. Another limitation is that the performance and robustness tests will be related to network convergence time.

Originally, the test system was meant to be a sustainable system which could be run for an infinite amount of time. However, this system can run for a maximum number of 71 minutes, due to the size of a standard 32 bit integer.

The processing power of an Arduino Uno is a major drawback to the current system. Since the Arduino only use a 16MHz processor, the maximum possible transmission speed was 2236 packets per second. To increase this speed significantly, a faster processor must be used. The processor must also be MEGA compatible in order to be able to run the W5100 Ethernet shield.

8.4 Future work

Due to the current calculation method of jitter, no clock synchronization was needed in this project. However, when adding more features to the traffic engine such as latency, a clock synchronization scheme must be added. This scheme can be implemented with a small effort using the GPIO pins of a raspberry and an Arduino. The thought of but not tested clock synchronization scheme looks as following:

Measuring the latency of a packet is a good way of testing the performance of the transmission time through the network. The functionality for calculating latency is implemented i.e. times-tamps.

Future work also include testing more traffic analyzing methods. Our ambition with the project was to implement one traffic analyzer for each traffic generator, i.e. using the USPI and Arduino based solutions as traffic analyzers as well. Due to lack of time, this was not an option. Even though using TShark was a very good method to measure the traffic, it cannot be used as a single solution. Instead the user need a traffic generator as well. An alternative could be implementing a traffic analyzer inside an Arduino, thus creating a unitary solution.

Testing how a solution developed on a real time operating system performs is also a part of possible future work. It was originally intended that some traffic generators and analyzers should
be developed on real time operating systems. This was planned in order to compare how well a real time operating system performs in comparison to the open source solutions available. This comparison could not be made because of the time restraints of the thesis.

Another solution to the traffic analyzer and traffic generator development would be to make one unit with both the analyzer and the generator. If the performance of the system could handle the workload of both the tasks, this could prove to be a way of calculating the latency. Since both the analyzer and the generator would be on the same device, it would require no time synchronization, however it would require an operating system possible of managing different tasks.

Future work could also include investigating the parameters which affects jitter. Why would both the maximum jitter and the average jitter increase at 125 PPS to then decrease again at 250? To investigate this issue further, more jitter readings should be made to find the exact turning points where the jitter increases and decreases.

Adding support for tests that can execute longer than 71 minutes can also be implemented. For this to work, the resetting of an integer has to be handled, else the sequences numbers and time stamps of a packet will be wrong.

In this thesis, there were only 2 protocols for networks in layer 2, and 2 protocols for layer 3. Future work include broadening this comparison, such as adding more protocols to compare. Perhaps changing the size of the network should be considered in order to see if the performance of the protocols are equal on larger or smaller networks. Implementing some kind of optimization such as the one mentioned by Basu and Riecke [2] for OSPF and comparing that to the normal OSPF is a potential future work.
References


A Appendix

A.1 FreeRTOS and USPI setup

FreeRTOS setup for Raspberry Pi requires a quite uncomfortable compiling process. First of all, the Raspberry Pi must be either model A+, B or B+, Raspberry Pi model 2 B+ and Raspberry Pi zero is not suitable for this project, as they run a different processor set. The original thought was to use Raspberry Pi model 2 B+, but this build failed as it requires a special type of .img file for the bootloader to understand it. The first step is to configure a Raspberry Pi compatible SD card to fat 32 format. This can easily be done using the standard way of implementing the raspbian operating system:

- `dd bs=4M if=2016-02-09-raspbian-jessie.img of=/dev/sdd`
- `dd bs=4M if=/dev/sdd of=from-sd-card.img`
- `truncate --reference 2016-02-09-raspbian-jessie.img from-sd-card.img`
- `diff -s from-sd-card.img 2016-02-09-raspbian-jessie.img`

By using this way, the full file structure of the raspbian operating system gets installed on the SD card. However, the only folder which is needed in this case is the boot folder, which will get automatically accessed on boot. The other folder can be deleted as it just contains the file structure for raspbian.

After this procedure, the freeRTOS Raspberry Pi port can be compiled. For compiling this library, firstly, an ARM cross compiler has to be installed: `sudo apt-get install arm-none-eabi`. When this is installed, the make file of freeRTOS has to be altered: `kernel.elf: LDFLAGS += -L "/usr/lib/gcc/arm-none-eabi/4.9.3" -lgcc`. This will produce a kernel.img file which is an executable operating system kernel that should be placed inside the boot folder of the Raspberry Pi card.

In order to enable Ethernet for the Raspberry Pi, the USPI library has to be installed. To install this, go in to the Rules.mk and change to the following options:

- `RASPi ?= 1`
- `PREFIX ?= arm-linux-gnueabihf-
- `ARCH ?= -march=armv6j -mtune=arm1176jzf-s -marm -mfloat-abi=hard`

After this has been done, the library can be compiled inside the /lib sub folder. It is possible of re-using code from one of the example projects in GIT, in this case we want to use the Ethernet example. However, the USPI library uses a conflicting type declaration of the size_t type. This conflict is cause by the arm-none-eabi cross compiler, hence this has to be removed in the uspienv/types.h. Simply comment out this type declaration for compiling the code.

A.2 PKTgen setup

Installing PKTgen-txrx was include the following.

1. `sudo apt-get install linux-headers-$((uname-r))`
2. `sudo apt-get install bc`
3. `git clone https://github.com/danieltt/pktgen.git`
4. `cd pktgen`
5. `make`

---

1[https://github.com/jameswalmsley/RaspberryPi-FreeRTOS]
2[https://github.com/rsta2/uspi]
6. insmod ./pktgen.ko
7. git clone https://github/jelaas/eth-affinity.git
8. cd eth-affinity
9. nano aff.c
10. add #include <stdint.h> to file
11. ./eth-affinity
12. sudo su
13. echo rx eth(x) > /proc/net/pktgen/pgrx

The following file was used to execute the 1ms second test with the Pktgen-tx-rx.

```bash
#!/bin/bash
function pgset() {
  local result
  echo 1 >PGDEV
}
# Config Start Here ———————————————————–
# thread config
CPUS=1
PKTS=`echo "scale=0; $3/$CPUS" | bc`
CLONE_SKB="clone_skb 1"
PKT_SIZE="pkt_size 60"
COUNT="count 100000"
DELAY = "delay 1000"
MAC="00:00:ac:bd:55:81"
ETH="eth0"
RATEP=`echo "scale=0; $1/$CPUS" | bc`
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
PGDEV=/proc/net/pktgen/pgctrl
pgset "add_device eth0"
pgset "$COUNT"
pset "$CLONE_SKB"
pgset "$PKT_SIZE"
pset "$DELAY"
pset "ratep $RATEP"
pset "dst 198.18.1.150"
pset "dst_mac $MAC"
pset "config 1"
pset "flows 1"
pset "flowlen 8"
PGDEV=/proc/net/pktgen/pgctrl
echo "Running... ctrl + C to stop"
pset "start"
```

"Done"
A.3 Running the TShark scripts

The following command will run TShark where it filters out all traffic from any source other than the IP 198.18.10.105, and it will remove ARP messages. It will run on the network interface eth0, the data from the reading is saved into test.pcap and the data from the IP packets payload is saved into testdata.txt.

```
sudo tshark -i eth0 -T fields -e data.data -w test.pcap src net 198.18.10.105 and not arp > testdata.txt
```

The following command will convert the file test.pcap to a readable test.txt file.

```
sudo tshark -r test.pcap > test.txt
```

A.4 Configurations used in the switches

The configurations used for the switches can be found in this section. These configurations were used during the tests when the protocols were compared.

A.4.1 FRNT configurations

The following text is copied from the configuration files used for the FRNT configurations on the switches.

```
# // Westermo WeOS v4.19.x, CLI Format v1.17
# RedFox RFIR-227-F4G-T7G-DC, art.no. 3641-4025-001 ser.no. 1131
aaa
  username admin hash $1$r6mXNVvD$JaDxe9xNk/MI7Ebdk7B0q.
end
system
  hostname redfox
  cpu-bandwidth-limit 1000
end
fdb
  mac 01:00:5e:00:00:01 port cpu, all
  mac 01:00:5e:00:00:02 port cpu, all
  mac 01:00:5e:00:00:04 port cpu, all
  mac 01:00:5e:00:00:05 port cpu, all
  mac 01:00:5e:00:00:06 port cpu, all
  mac 01:00:5e:00:00:09 port cpu, all
  mac 01:00:5e:00:00:0a port cpu, all
  mac 01:00:5e:00:00:0d port cpu, all
  mac 01:00:5e:00:00:0e port cpu, all
  mac 01:00:5e:00:00:12 port cpu, all
  mac 01:00:5e:00:00:18 port cpu, all
  mac 01:00:5e:00:00:66 port cpu, all
  mac 01:00:5e:00:00:6b port cpu, all
end
alarm
  trigger 1 frnt
  ring 1
  severity active warning inactive notice
  condition high
  action 1
end
action 1
target snmp log led digout
```
A.4.2 RSTP configurations

The following text is copied from the configuration files used for the RSTP configurations on the switches.

```
# \ Westermo WeOS v4.19.x, CLI Format v1.17
# RedFox RFIR-227-F4G-T7G-DC, art.no. 3641-4025-001 ser.no. 1131
aaa
username admin hash $1$r6mXNVvD$JaDxe9xNk/Ml7Ebdk7B0q.
end
system
hostname redfox
cpu-bandwidth-limit 1000
end
fdb
mac 01:00:5e:00:00:01 port cpu, all
mac 01:00:5e:00:00:02 port cpu, all
mac 01:00:5e:00:00:04 port cpu, all
mac 01:00:5e:00:00:05 port cpu, all
mac 01:00:5e:00:00:06 port cpu, all
mac 01:00:5e:00:00:09 port cpu, all
mac 01:00:5e:00:00:0a port cpu, all
mac 01:00:5e:00:00:0d port cpu, all
mac 01:00:5e:00:00:0e port cpu, all
mac 01:00:5e:00:00:12 port cpu, all
mac 01:00:5e:00:00:18 port cpu, all
mac 01:00:5e:00:00:66 port cpu, all
mac 01:00:5e:00:00:6b port cpu, all
mac 01:00:5e:00:00:fb port cpu, all
```
A.4.3 RIP configurations

The following text is copied from the configuration files used for the RIP configurations on the switches.

```bash
# Westermo WeOS v4.19.x, CLI Format v1.17
# RedFox RFIR-227-F4G-T7G-DC, art.no. 3641-4025-001 ser.no. 1131
aaa
  username admin hash $1$r6mXNVvD$JaDxe9xNk/MI7Ebdk7B0q.
end
system
  hostname redfox
end
```
fdb
mac 01:00:5e:00:00:01 port cpu, all
mac 01:00:5e:00:00:02 port cpu, all
mac 01:00:5e:00:00:04 port cpu, all
mac 01:00:5e:00:00:05 port cpu, all
mac 01:00:5e:00:00:06 port cpu, all
mac 01:00:5e:00:00:09 port cpu, all
mac 01:00:5e:00:00:0a port cpu, all
mac 01:00:5e:00:00:0d port cpu, all
mac 01:00:5e:00:00:0e port cpu, all
mac 01:00:5e:00:00:12 port cpu, all
mac 01:00:5e:00:00:18 port cpu, all
mac 01:00:5e:00:00:66 port cpu, all
mac 01:00:5e:00:00:6b port cpu, all
mac 01:00:5e:00:00:fb port cpu, all
end
alarm
toggle 1 frnt
ring 1
severity active warning inactive notice
condition high
action 1
end
action 1
target snmp log led digout
end
end
port ALL
speed-duplex auto
end
no spanning-tree
vlan 1
name vlan1
untagged 1-3,6-27
no igmp
end
vlan 2
name vlan2
untagged 5
end
vlan 3
name vlan3
untagged 4
end
iface vlan1 inet static
distance 1
primary
management ssh http https ipconfig snmp
address 198.18.1.11/24
end
iface vlan2 inet static
distance 16
management ssh http https ipconfig snmp
address 198.18.2.2/24
end
iface vlan3 inet static
distance 16
management ssh http https ipconfig snmp
address 198.18.3.1/24
end
router
rip
network 198.18.2.0/24
network 198.18.3.0/24
end
end
no snmp-server
ntp
end
no ipconfig
no lldp
no web

A.4.4 OSPF configurations

The following text is copied from the configuration files used for the OSPF configurations on the switches.

# \ Westermo WeOS v4.19.x, CLI Format v1.17
# RedFox RFIR-227-F4G-T7G-DC, art.no. 3641-4025-001 ser.no. 1131
aaa
username admin hash $1$r6mXNVvD$JaDxe9xNk/MI7Ebdk7B0q.
end
system
hostname redfox
end
fdb
mac 01:00:5e:00:00:01 port cpu, all
mac 01:00:5e:00:00:02 port cpu, all
mac 01:00:5e:00:00:04 port cpu, all
mac 01:00:5e:00:00:05 port cpu, all
mac 01:00:5e:00:00:06 port cpu, all
mac 01:00:5e:00:00:09 port cpu, all
mac 01:00:5e:00:00:0a port cpu, all
mac 01:00:5e:00:00:0d port cpu, all
mac 01:00:5e:00:00:0e port cpu, all
mac 01:00:5e:00:00:12 port cpu, all
mac 01:00:5e:00:00:18 port cpu, all
mac 01:00:5e:00:00:66 port cpu, all
mac 01:00:5e:00:00:6b port cpu, all
mac 01:00:5e:00:00:fb port cpu, all
end
alarm
trigger 1 frnt
ring 1
severity active warning inactive notice
condition high
action 1
end
action 1
target snmp log led digout
end
end
```plaintext
port ALL
speed-duplex auto
end
no spanning-tree
vlan 1
name vlan1
untagged 1-3,6-27
no igmp
end
vlan 2
name vlan2
untagged 5
end
vlan 3
name vlan3
untagged 4
end
iface vlan1 inet static
distance 1
primary
management ssh http https ipconfig snmp
address 198.18.1.11/24
end
iface vlan2 inet static
distance 16
management ssh http https ipconfig snmp
address 198.18.2.2/24
end
iface vlan3 inet static
distance 16
management ssh http https ipconfig snmp
address 198.18.3.1/24
end
router
ospf
network 198.18.2.0/24 area 0.0.0.0
network 198.18.3.0/24 area 0.0.0.0
end
derouter
snmp
no snmp-server
ntp
end
no ipconfig
no lldp
no web
```

### A.5 Test results

Figure 22, Figure 23, Figure 24, Figure 25, and Figure 26 are the individual results from each measurement which was summarized in Figure 13.
Figure 22: Jitter generated from the arduino

Figure 23: Mausezahn jitter
Figure 24: USPI jitter

Figure 25: PKTgen jitter
Figure 26: Mausezahn RT preempt jitter