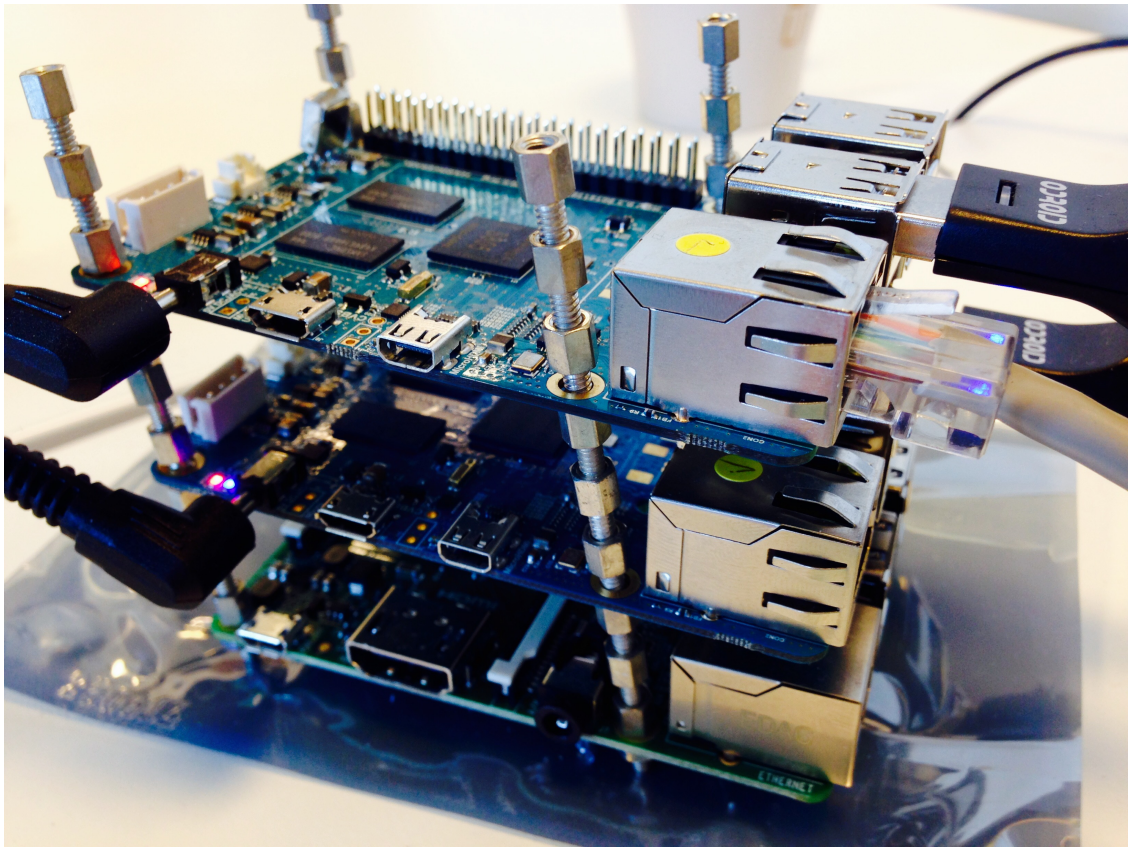

Prototyping a network traffic tester using a distributed system of single board computers.

Master's thesis in Computer science and engineering

SIMON DIRNBERGER

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Prototyping a network traffic tester using a distributed system of single board computers.

SIMON DIRNBERGER

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Prototyping a network traffic tester using a distributed system of single board computers.
SIMON DIRNBERGER

Supervisor: Olaf Landsiedel, Computer Science and Engineering.
Examiner: Marina Papatriantafilou, Computer Science and Engineering.

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Picture of a stack of the single-board computers used in the actual prototype of the system developed in this thesis work.

Typeset in LaTeX
Gothenburg, Sweden 2021

Prototyping a network traffic tester using a distributed system of single board computers.
SIMON DIRNBERGER
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Network traffic testers are integral to developing and evaluating hardware that is used for the ever-increasing demand for faster, better and more covering data networks. However the testers typically used come with a hefty price tag and are sold by companies that specialise in the field. This paper tries to prove whether a distributed system of smaller and cheaper single-board computers can be built to function as a high-performant network traffic tester. We look at similar solutions for network traffic testers, both proprietary and non-proprietary. We formulate a set of requirements, both hardware and software, and in this paper suggest a design, implementation and evaluation of such a system. The result of our findings is a working traffic tester built from single-board computers running Linux and using the the kernel module pktgen for traffic generation, netsniff-ng for capturing traffic and the framework MPICH for communication. The test results are promising when it comes to traffic rate and accuracy, given certain limitations in the system. However the increased complexity in its usage makes it unwieldy to use and requires more work in order for it to be a viable option. However, given enough time for it to become more user-friendly we believe that this solution can compete with the more expensive alternatives.

Keywords: single-board computers, traffic generator, data network, distributed systems, OSI

Acknowledgements

This thesis has been a long project. Not in the sense that I've been working on it continuously for a long time. More just for the fact that it started up 6 years ago and then never quite got finished... Until now! It's been a long road and I never thought I'd actually go ahead and finish this after the break I had. It only took 6 years for me to pick it up again. For this I'd very much like to thank my beautiful wife Emma for pushing me and enabling me to starting it all up again. I'd like to thank my incredibly patient, kind and caring supervisor Olaf Landsiedel, he's even left Sweden because of me! Joke's aside. I very much appreciate it Olaf, for giving me this chance. I'd also like to thank Vincenzo Gulisano, the head of the department of MPCSN, and Mariana Papatriantafilou, the examiner, for the time out of your precious schedules and helping this lost thesis writer in the process of finishing his thesis.

I'd also like to thank mom, dad, my little sister and her partner Daniel - for the warm support and the friendly cheering! And a special shoutout to my little sister, she's giving birth in just a few weeks! You are the best, Fia!

I'd like to thank my employers at Layer 10 (Martin, Mattias, and Jimmy) for giving me some time off work to properly tackle this project. And also for taking me on, and this thesis, all those years ago. Who would've thought I'd be here now and actually finishing this?

Also a final thank you to my former colleague and supervisor at Layer 10, Henrik Almegren, who was the person that initially came up with the idea for this thesis and helped me with all the practicalities.

Simon Dirnberger, Kungälv, November 2021

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem formulation . . . 2
1.3 Thesis outline . . . 2

2 Background **3**
2.1 The difficulties of a distributed system . . . 3
2.1.1 Causality . . . 3
2.1.2 Consistency . . . 4
2.1.3 Timing . . . 4
2.2 Network packets handled in the Linux kernel . . . 5
2.3 Communication between the nodes, message passing is our virtue . . . 6
2.3.1 The Message Passing Interface . . . 7
2.3.1.1 Addressing in message passing . . . 8
2.3.1.2 Communication flow . . . 8
2.3.1.3 How to communicate between participants . . . 9
2.3.2 MPICH - an MPI implementation . . . 9
2.3.2.1 MPI4Py - A Python package for MPI . . . 9

3 Related work **11**
3.1 Open source . . . 11
3.1.1 Non-distributed . . . 11
3.1.2 Distributed . . . 12
3.2 Proprietary . . . 13
3.2.1 Non-distributed . . . 13
3.2.2 Distributed . . . 14
3.3 Summary . . . 14

4 Design **15**
4.1 Design requirements . . . 15
4.2 System overview . . . 16
4.3 The two traffic flows . . . 16
4.3.1 Control and communication network . . . 17

4.3.2 Traffic network . . . 18
4.4 The master node . . . 19
4.4.1 Software component . . . 19
4.4.1.1 Grouping into transmitting and receiving nodes . . . 19
4.4.1.2 Distributing the tasks of sending and receiving traffic 20
4.4.1.3 Synchronising the groups . . . 21
4.4.1.4 Collect the results of the sent traffic . . . 22
4.4.2 Hardware component . . . 23
4.5 The transmitting nodes . . . 23
4.5.1 Software component . . . 24
4.5.2 Hardware component . . . 24
4.6 The receiving nodes . . . 25
4.6.1 Software component . . . 25
4.6.2 Hardware component . . . 26
4.7 Packet generation - the transmitting task . . . 26
4.8 Packet capturing - the receiving task . . . 27
4.9 Summary . . . 28

5 Implementation **29**
5.1 System overview . . . 29
5.2 Operation . . . 29
5.3 Master node . . . 30
5.4 Receiving nodes . . . 32
5.5 Transmitting nodes . . . 34
5.6 The hardware . . . 36
5.6.1 The single-board computer - ODROID-C1 . . . 37
5.6.2 The aggregating switch - Ericsson SP 210 . . . 38
5.6.3 Pre-study of the hardware . . . 38

6 Evaluation **39**
6.1 Packet generation . . . 39
6.1.1 Test methodology . . . 39
6.1.2 The test scenarios . . . 39
6.1.3 SBC hardware evaluation - finding the hard limit . . . 44
6.2 Traffic configuration . . . 45
6.2.1 Criterias . . . 45
6.2.2 Scapy/tcpreplay . . . 46
6.2.3 Pktgen . . . 46
6.2.4 tcpreplay . . . 47
6.3 Packet capturing . . . 48
6.3.1 Test methodology . . . 48
6.3.2 The test scenarios - finding the zero packet drop rate . . . 49
6.4 Bottlenecks . . . 50

7 Conclusion **53**

Bibliography **55**

A Appendix A **I**

B Appendix B **V**

List of Figures

2.1 A limited figure of the Linux kernel network flow. 7

4.1 An overview of the system design. 17
4.2 An overview of how the control traffic and communication flow between master node and the groups of transmitting and receiving nodes. 18
4.3 An overview of how the master node splits the provided node addresses into each transmitting and receiving group respectively. 20
4.4 Process of synchronising the start of a transmission. 22
4.5 Process of synchronising the end of a transmission. 22

5.1 Sequence diagram of a typical system execution. 30
5.2 Software execution flow from master node's point of view 31
5.3 Flow chart of a receiving node. 33
5.4 Flow chart of a transmit node. 35
5.5 A relations diagram. 36

6.1 The test setup. 40
6.2 Plotted traffic results from scenario 1 to 3 41
6.3 Plotted traffic results from scenario 4 to 6 42
6.4 Plotted traffic results from scenario 7 to 9 43
6.5 Actual outputs of the software with a given packet rate (pps). 47
6.6 The capturing test setup. 48
6.7 Packet rates and the percentage of failed tests. 50

List of Tables

5.1 Comparison of different single-board computers. 38

6.1 Different test scenarios 40
6.2 Finding the k with 30 000 pkts/s 41
6.3 Finding the k with 60 000 pkts/s 42
6.4 Finding the k with 110 000 pkts/s 42
6.5 Conversion to Mpbs 43
6.6 Performance test, comparison between Raspberry Pi and Banana Pro. 44
6.7 Performance test, comparison between Odroid and Banana Pro. . . . 45
6.8 Scenario 1: Large packet size - high throughput 49
6.9 Scenario 2: Low packet size - high packet rate 49

B.1 Table showing all tests made to find the zero packet drop rate. VII

1
Introduction

1.1 Motivation

Single-board computers (SBC), such as Raspberry Pi, BeagleBone and ODROID, are increasing in popularity as new ideas and projects are shared in the thriving SBC communities. Systems such as simple web servers, media centers, monitoring systems [1] and even super computers [2] have been implemented with the popular tiny computers. The SBCs display high versatility despite their limited hardware.

Layer 10 is a company that use proprietary hardware Ethernet testing products, hereby referred to as testers, for testing microwave radio products. The tester generates and analyses network traffic at high transfer rates. A typical test case would be to define a stream of packets that contain specific data. The data is then sent with different intervals from one port of the tester, through the microwave radio unit and further into the other port of the tester where it is made sure that all packets sent are also received and in no way corrupt or altered. The analysis consists of checking that all the fields set in receiving end coincide with the packets that were sent from the transmitting side.

The problem is that the testers are typically very expensive. As of late there is also an increase in demand for the testers as other projects have started that are in need of performing the same kind of tests. However instead of spending a lot of resources on new hardware, the company nurtured the idea of trying to design and implement a new cost-efficient tester that is based on open source software and cheaper hardware.

This is where the company wants to put SBCs to use. As the requirements of the transfer rates are high (up to 1 Gbps) typically a single SBC is not enough. However if the combined performance of a distributed system of SBCs, supported with open source software, could match the functionality of a proprietary tester it would be of high interest to use this kind of system alongside, or instead of, the proprietary testers. If we can utilise the scalability of a distributed system, we can theoretically increase the performance indefinitely, given that performance increase linearly with the number of nodes added to the system. For instance, let us say we reach a traffic rate of 1 Gbps at five nodes. It would then be interesting to explore whether we can achieve 2 Gbps by adding five more nodes to the system. Then the distributed system of SBCs becomes very attractive as the cost of another node is typically way cheaper than another proprietary tester. Thus the problem becomes how to design and implement a distributed system of SBCs that can generate and analyse network traffic at high speeds.

1.2 Problem formulation

The thesis can be split up into several problems or issues that need to be investigated. The following list summarises these problems.

- Investigate the testing suites of the proprietary systems and pinpointing which aspects of the traffic generation and analyses are the most important.
- Evaluation of which hardware that suits the system best. The evaluation is made of weighing the different hardware's performance against its price.
- Evaluation of which open source software for packet generation and analysis that meet the requirements of Layer 10's testing suites. In the evaluation it is also important to investigate which software utilises the limited hardware performance of the SBCs the best.
- Designing and implementing a distributed system that distributes the task of packet generation and analysis among the participating nodes of the system in a synchronised manner.
- Evaluating the performance of the system and compare it to the proprietary products.

1.3 Thesis outline

The thesis is split up into several chapters. Chapter 2 presents the background of which this thesis is based on. It goes through the concepts and details of the work that this thesis uses as its foundation. Reading through this chapter is optional, it might be of interest if some concepts of the thesis are unfamiliar or does not make any sense. Chapter 3 summarises the work that is related to this thesis. It is split into several sections of proprietary, non-proprietary, distributed and non-distributed solutions of packet generators and analysers that does similar things to what the work of this thesis is trying to achieve. Chapter 4 goes into detail of designing the system. It covers how the packet generation and analysis is defined as distributed tasks. We introduce the concepts of receiving and transmitting groups of nodes, and the master node that works as the handler of the system. Chapter 5 consists of the actual implementation of the system. That is how the system is interacted with and presents details of how the software is built together to form the distributed packet generator and analyser. Chapter 6 exhibits the evaluation results of the system. It covers the most important evaluations made in establishing how good, or bad, different aspects of the system worked. These evaluations include everything from which SBC would suit best for the thesis to how the final system stood against the proprietary tester. In Chapter 7 we present the conclusions that could be drawn from the work and results of the evaluations. The chapter also presents thoughts about what improvements and changes that could be done in the future.

2
Background

In this chapter we look into the background material that lay as a foundation for the thesis. We go into detail of the established concepts and methods that are used within the thesis work, i.e. the work which the thesis is based upon. The chapter is split into sections where we present information related to the basic problems of a distributed system. In Section 2.1 we look at common difficulties when building a distributed system. Each subsection details a specific problem. Section 2.2 describes how the Linux kernel handles network packets, as this is of importance when we want to utilise as much as possible from the limited hardware. Section 2.3 describes how the communication is handled between the nodes and how MPI is used to pass information between the participating nodes of the distributed system.

2.1 The difficulties of a distributed system

This section touches upon three major problems related to implementing distributed systems. In the Section 2.1.1 we go through how causality is an issue in distributed systems. In Section 2.1.2 the implications of consistency between the nodes is explained and why it is important that every node that share data should have a common view of the system. Section 2.1.3 shines light on the problem of timing, why each node needs to be synchronised.

2.1.1 Causality

Causality is one of the challenges in modelling distributed systems. The problem stems from a property that comes with every distributed system. The property is that in distributed systems there is no way of being sure when messages arrive at participating nodes, or even if they arrive at all. Because of this property there is no safe way for participants of a distributed system to agree on what, or in what order, events happen. Since distributed systems do not share clocks there is no way to agree on a time, and then there is no possibility to agree on when events happen.

The solution for this problem is to not use time as a means for knowing when events happen, but instead using logical time. At every event we increase our logical clock by one step. The problem comes when we want to know if one event on one participating node happened before another event on another node. This is where causal ordering comes in. The only way to know for sure if one event on one node happened before another event on another node is to find an instance where these two events are connected to each other.

On a set of machines we have two different types of events, events that only occur on a single machine and events that occur between machines. There are three rules that describe how causal ordering works.

1. If event A happens before B on a local machine then A also happens before B in the global order. Notated like this $A \rightarrow B$.
2. Sending a message m always happens before receiving that message, notation is $send(m) \rightarrow recv(m)$.
3. Causal event ordering is also transitive, if $A \rightarrow B$ and $B \rightarrow C$ then it is also true that $A \rightarrow C$.

2.1.2 Consistency

In a distributed system where we share a common mutable state one of the main problems is keeping that state consistent over all different participating machines. When that state is replicated over all machines we have an issue where different updates can happen on different machines at different times. We can only define a distributed system as consistent when this replicated state only exists in one form. In other words, there can never be a replicated state that looks different depending on from which participating machine you access this state from.

To solve this issue there are different consistency models that can be implemented in a distributed system. They all balance between delivering a loose consistency or a strict consistency in a distributed system. Loose consistency means that the system will be effective but the state might not be updated at all machines. Strict consistency is inefficient but ensures that the state is always up-to-date at all machines.

In this thesis work the problem is avoided because the system is modeled so that data sent from each node is unique to that node. Furthermore the task of sending and receiving data is isolated. No operations or states are really shared between the nodes. Thus there is no real need for a consistency model. Each node operates on its own state.

2.1.3 Timing

In addition to keeping track of logical clocks and event ordering we also need to keep track of the actual time in a distributed system. We need to keep a consistent time of day across the whole system. In other words we need a way to synchronise timing between machines in a distributed system. To put it in context for this thesis we need to find a way for the processes to agree on when to start sending data, otherwise we have no way of ensuring a packet rate or when and how the data will be sent.

As [3] discusses there are a few synchronisation algorithms in distributed systems for machines to agree on a common time. Some of them rely on synchronising clocks from another machine that has a reliable time, a master node so to speak. The algorithm then also takes delay into consideration by using error bounds. In this thesis we rely on a construct called synchronisation barrier, it is one of the collective calls specified in the Message Passing Interface (MPI) that will be discussed in detail in a Section 2.3.1. In this call the master node initiates a barrier and does

not let the state machine to continue executing before all participating nodes have reached this barrier. This is another way of synchronising the timing between the nodes and the method is relied upon in the work of this thesis.

2.2 Network packets handled in the Linux kernel

This section touches upon why it is important how the Linux kernel handles network traffic. Trying to limit the overhead and getting as close to the hardware as possible can significantly increase the performance and make the system more accurate. This section features explanations and figures to describe the working of the network data handling of the Linux kernel.

We will focus on the communication layers which are relevant for this thesis. This means that we will limit the network data handling up to what is called the session layer in the OSI model, which is the entry level for user space over to kernel space. It makes sense to do this explanation through the transmission of a packet, from user space to kernel space and then to the actual network interface card (NIC).

The transmission of an Ethernet packet over IPv4, that is not the configuring or creating of it, starts at Layer 5 - the session layer. In the session layer there are three system calls that can be used to send data over the network. They are called `write`, `sendto` and `sendmsg`. The `write` call sends memory data to a file descriptor, `sendto` sends memory data to a socket and `sendmsg` takes a composite message and sends it to a socket. Note that each of these are system calls, or kernel entry points, which means that they are interrupts to the kernel. What this means is that data must be copied back and forth between user and kernel space when these system calls are made, wasting processor cycles on simply copying data. This because kernel space and user space are running on separate address spaces.

In Layer 4, called the transport layer, there is a function called `tcp_sendmsg` that is used for sending each segment of a message. Before we delve into the details of how that function operates let us take a look at what an `sk_buff`, socket buffer, is and what it does. An `sk_buff` is simply a data structure that the kernel uses for all network-related queues or buffers. It is implemented as a doubly linked list and the structure contains all control information required for a packet. The network data is managed through the `sk_buff` data structure. The `tcp_sendmsg` uses this data structure in its operations which is why some basic explanation is needed. Now the `tcp_sendmsg` function operates in four steps.

1. Find an `sk_buff` with space available.
2. Copy data from user space to the `sk_buff` data space, notice that this is again a call from user to kernel space. As [4] discusses the data duplication between kernel and user buffer is not really necessary, some of it can be deleted to decrease overhead.
    - In this step the buffer space for the `sk_buff` is pre-allocated. If it ever runs out of space the communication will stop and the data will stay in the user space until `sk_buff` data space is available again.
    - The size of an `sk_buff` is the same as the Maximum Segment Space (MSS) + header length.

- This is where the TCP segments are defined, all data that ends up in the same `sk_buff` becomes a TCP segment.

3. The TCP queue is activated and function `tcp_transmit_skb` is called to send the packets. Note that skb stands for `sk_buff`.
4. `tcp_transmit_skb` builds the TCP header and then clones the `sk_buff` to send it all to the network layer where the kernel will continue processing the packet. The network layer is called through a virtual function called `queue_xmit`.

At Layer 3 - the network layer, the process continues with the following steps:

5. `queue_xmit` does any necessary routing and then creates the IP header.
6. At this point there are a series of kernel space methods that are called in order to handle problems such as network filtering, outputting data in accordance with destination ip address and fragmentation.
7. The most important virtual method at this stage is the `ip_output` that takes care of finally sending it to the output device. In other words which entry in the network devices that shall be used.
8. To perform network filtering, e.g. NAT and firewalls a hook named `nf_hook` is called to modify or discard the packet accordingly.
9. The previous routing decisions gives us a destination object. In this object a model of the reciever's IP address is stored. A virtual method called `dst_entry` is used to to perform the output of the data.

When the packet finally arrives at Layer 2 the kernel's work is to schedule the packets that shall be sent. It puts the `sk_buff` on the device queue with a virtual method called `dev_queue_xmit` and eventually the `sk_buff` is sent and removed from the queue, if the sending fails it will be queued again. The device driver talks to the network device, the virtual method `dev_hard_start_xmit` is responsible for talking to the device, for scheduling transfer of the `sk_buff` and when the network device replies that it is done the buffer will be freed.To help us understand this flow we can look at a limited figure of the kernel network flow presented in Figure 2.1.

In summary the transmission of a packet is typically initiated by a system call where there is a lot of moving of data between user and kernel space. It then uses a set of kernel methods to deal with the packet down through the communication layers and eventually speaks with the device driver to send the packet out on the network device.

2.3 Communication between the nodes, message passing is our virtue

This section explains the practice of message passing and why it is important for this thesis work. We also take a look at the Message Passing Interface (MPI) and how it defines a solid specification of how message passing should be implemented. Furthermore the section details the different constructs that are defined within MPI that the developed system uses to function properly.

One way of explaining message passing is done through looking at why it is needed. Traditional computing involves a single processor handling all calculations

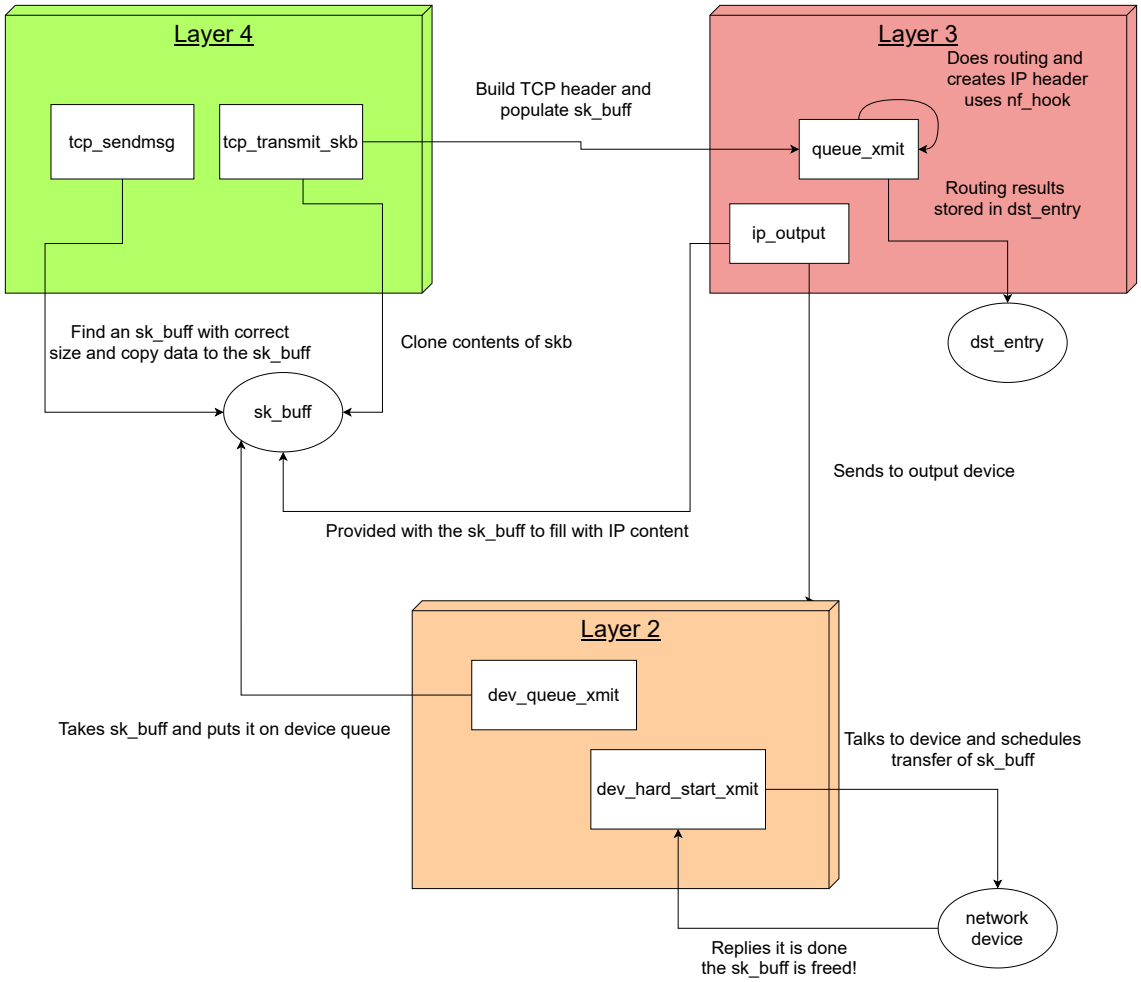

Figure 2.1: A limited figure of the Linux kernel network flow.

and operations using a single address space. It has no one to care about except for itself and its own state. However in this thesis we are looking at how to distribute the tasks over several processors - distributed processing. We are executing concurrent calculations on different processors, each processors has its own state and calculations. The processors do not share a common address space so they cannot communicate via shared variables. This is where message passing comes in, it is used for the communication by sending and receiving messages between the participating processors to achieve the results of the distributed calculations.

2.3.1 The Message Passing Interface

MPI is not an implementation, it is a library interface specification for message passing. MPI defines a model for message-passing parallel programming. The model, or standard, was defined by a collective of parallel computing vendors, computer scientists and application vendors [5]. The model specifies how data is moved from one process to another and the cooperative operations that are needed for each process to accomplish this. In addition to the plain message-passing between processes other functions are also provided such as collective operations, remote access operations,

dynamic process creation and parallell I/O.

All these operations are expressed as functions or methods according to the appropriate language bindings. Again, MPI is not an implementation it is a specification on which methods and operations are needed to provide a solid working message-passing standard. The goal of MPI is to develop a well-defined standard to write message-passing programs. It should be efficient, portable, practical and flexible to make it widely used to writing message-passing programs.

There are quite a few different approaches when using message passing as a means of communication. There are different ways of addressing participators, different ways of how the communication flow is handled and also different ways of how the communication is done. In the following subsections we will look at each approach and also which approach that was used for this thesis work. In subsection 2.3.1.1 the different approaches on how to handle addressing of participating nodes is explained. Subsection 2.3.1.2 presents the different flows of communication that can be utilised in a message passing context. In the final subsection 2.3.1.3 we look at the two preferred ways of how the communication is done between the nodes.

2.3.1.1 Addressing in message passing

When addressing nodes there are mainly two approaches, direct and indirect addressing. Direct addressing lets each processor have its own address, or name, and thus a message is directly sent, or received, by one processor. Indirect addressing names instead a channel, or a group, that is addressed for receiving or sending messages. Every processor that is participating in the channel receives every message that is sent to it. Indirect addressing is also the solution that was used for this thesis work.

In the context of this thesis this essentially means that each SBC that participates in sending, or receiving, of data traffic belongs to a group. In MPI terms this is called a communicator. So the group of receiving SBCs shares a communicator that lets the receiving SBCs message each other. Respectively, the group of sending SBCs has another communicator that handles the communication between the nodes. Furthermore each SBC is uniquely identified within each communicator by a rank. The rank is simply a unique number, however that number is only unique within each communicator. Applied to our scenario this means that ranks can uniquely identify senders within the sender communicator but does not carry over to the receivers because they belong to another communicator.

2.3.1.2 Communication flow

When it comes to handling communication flow in a message passing system there are a few different ways of doing it. A communication flow can either be bidirectional or unidirectional. When it is unidirectional messages only travel in one direction at a given time. If it is bidirectional messages can travel in either direction at a given time.

In addition to these flows MPI also defines something called collective calls. They are specific calls that makes it easier to handle the communication flow in a group by letting the communicator handle the flow. In the work of this thesis these

collective calls were used to establish a unidirectional communication flow between the SBCs.

2.3.1.3 How to communicate between participants

Lastly a communication between processors can be based upon yet another two different principles. Either it can be based upon synchronous or asynchronous communication. In asynchronous messaging a message can be sent and then the processor can continue executing, the send operations are non-blocking. Essentially this means that the sending processor can execute for an arbitrary amount of time and message delivery is not guaranteed. In synchronous messaging the sender blocks until the message has been received. In this thesis the communication is synchronous, even additional constructs are used to make sure that every receiving processor has reached a certain state before every processor can start executing again.

2.3.2 MPICH - an MPI implementation

In this section we present one of the actual implementations of MPI. This implementation is called MPICH and it is this framework that is used for calling all the functions specified in MPI. With this section we look at what MPICH is and its usages.

MPICH is a high-performance and widely portable open-source implementation of MPI[6]. In addition to providing implementations for all MPI calls it is also used to compile and execute MPI programs. Executing MPI programs with MPICH lets you define how many processes should be run and on what nodes they should be run on. In this thesis work MPICH has been used to distribute the tasks of transmitting and receiving network traffic.

2.3.2.1 MPI4Py - A Python package for MPI

MPI4Py is a package for Python that enables us to develop code that integrates towards MPI implementations. Python is a programming language that allows us to use efficient data structures to develop high-level solutions [7]. MPI4Py links up the code to the MPICH libraries and helps in not having to develop any parts of the solution in C. By keeping the program code in a high-level language we can limit the scope of the thesis and avoid potential complications. However it is important to be aware of the consequences this might have on the low-level kernel operations, e.g. avoiding high-cost function calls in Python and try and separating the code that is calling on the kernel space applications.

3
Related work

In this chapter we look at similar work and solutions that are related to what this thesis aims to achieve. The chapter is divided into four sections where each section describes related work out of four different categories. The four categories are distributed and non-distributed proprietary packet generators and analysers and corresponding open-source alternatives.

3.1 Open source

This section describes the open-source packet generator and analysers. These are all software that can be installed on any PC that fulfills the prerequisites. As such they are often general purpose testers and thus are not as highly performant as the proprietary solutions. The functionality is built by software and often relies on the `libpcap` library for Linux. The functionality is built above the hardware and not dependent on what kind of hardware that runs the kernel, this is also why the performance is limited.

3.1.1 Non-distributed

Tcpdump

tcpdump is a terminal-based packet analyser. It uses the *libpcap* library to capture packets that are transmitted or received via the network interface. It prints the contents of the packets captured according to the filter that is supplied to the application at start [8]. It is a versatile tool for analysing TCP/IP network traffic that comes through the host which *tcpdump* is running on.

Tcpreplay

Tcpreplay is a suite of tools for replaying packets that have been previously captured and, usually, stored in a pcap file. A pcap file is simply a file format for storing packet data from a network. It is also possible to rewrite Layer 2, 3 and 4 headers of the captured packet and then send the altered packet. According to the application's website the goal of the tool is to enable reliable and repeatable means for testing the functionality of network devices [9].

Iperf

Iperf is a tool to measure the performance of a network. It creates traffic streams and do performance measurements with these streams. It is first and foremost a performance measuring tool that can detect packet loss. However it can only be used to measure TCP/UDP traffic and the protocols at these layers [10]. For the master thesis the traffic needs to be measured and analysed at lower levels in the OSI model, already in Layer 2.

Ostinato

Ostinato is an open-source software application that can craft and analyse packets [11]. It can be installed on both Windows, Linux, BSD and Mac OS X and features a GUI for crafting and sending packets. It is also possible to configure several streams with different protocols and different speeds. It supports most standard protocols and the user can modify any field of any protocol, it is also possible to craft malformed packets which is of relevance to testing the radio products. The streams can be configured to send in different patterns according to a specified rate, bursts or number of packets. At the time of writing this thesis it does not have an API for extending or using the software programmatically. However according to their website they plan to implement this.

3.1.2 Distributed

D-ITG

Distributed Internet Traffic Generator (D-ITG) is a platform that is developed by a group belonging to the Department of Electrical Engineering and Information Technologies at the University of Napoli Federico II [12]. It can simulate IPv4 and IPv6 traffic by replicating workload of Internet applications. In addition it can perform measurement tests of throughput, delay, jitter, and packet loss. By being of a multi-threaded design it is possible to set up multiple parallel traffic flows going from sources to destinations, similar to what this thesis wants to achieve. The platform has many features which are listed on the website. However it is not possible to craft packets of any form, but rather of the forms supported by D-ITG. The platform does not provide any fault tolerance or any form of synchronisation.

DiCAP

DiCAP, or Distributed Packet Capturing, is a scalable architecture and implementation for distributing packet capturing proposed by a group of researchers at the University of Zürich [13]. In the paper an evaluation is made of the DiCAP implementation and shows that it can perform loss-less IP packet header capture at high speeds. However the architecture is only for packet capturing and not packet crafting or packet generation.

3.2 Proprietary

This section describes the different proprietary packet generators and analysers. What characterises these hardware proprietary solutions is that they are usually very expensive, a result of the products being meticulously developed for a specific purpose. In this case being end-to-end Ethernet testing. What the companies that produce these products can do is to place the functionality close to the hardware and thus obtain highly performant and deterministic products.

3.2.1 Non-distributed

These traffic testers are all built into a piece of hardware that is sold by the companies. Some offer distributed solutions but these are used for very specific purposes and not for the general case. What unites each company is that most of their products are hardware instruments that provide high performance testing through the instrument.

Anritsu

Anritsu [14] is a company that produces many different types of analysers and generators. Layer 10 has one of Anritsu's Ethernet/IP network data analysers which is used in testing the radio products today. The analysers provide wire speed performance analysis and packet generation within the same product. Multiple protocols can be decoded and displayed for captured data up to 10 Gbps, and it is also possible to do so simultaneously on multiple channels containing different streams of data. It uses modular plug-in units to support performance, jitter, and EoS (Ethernet over SDH/Sonet) measurements of networks. Supporting both Ethernet and IP technologies various applications such as QoS and IPTV streaming services can be tested.

The tester that is used in the Layer 10 lab is the MP1590B which is capable of measuring IP networks up to 10 Gbit. The product comes with a customised Windows installation for control of the instrument. It can be both controlled remotely or through a small screen on the actual instrument. The instrument supports many types of functions. However many of the functions are not included in the instrument but need separate modules that is installed to the instrument. For example to measure traffic up to 10 Gbit a special module is needed that is sold separately. The instrument supports multichannel measurement which means it can parallelise both generating and capturing traffic, however not indefinitely as it can only be done over the number of physical ports on the instrument. Both packet contents and the packet stream are fully customisable. The user has many different options and settings to utilise and almost any IP protocol can be tested with this instrument.

Spirent

Spirent [15] is another company that much like Anritsu delivers Ethernet testing solutions. They deliver many kinds of products and have different solutions depending on what the customer intends to test. They supply both hardware and

software for testing Ethernet networks up to 400 Gb/s. Spirent also offers solutions to achieve many different kinds of validations. For example Layer 1-7 validation on both Ethernet and optics medium, IP/MPLS protocol validation, Carrier Ethernet meaning validation of fault and performance procedures, mobile backhaul timing and synchronisation and many other features.

Spirent TestCenter is an end-to-end product that provides measurement solutions for several types of testing, most importantly performance testing. The TestCenter supports stateful Layer 2-7 traffic generation and analysis up to 100Gbit/s Ethernet and fibre channels. Through the TestCenter's software it is possible to use wizards to configure any type of packet and interface settings. There are also pre-defined test cases for testing specific protocols and functions. It features comprehensive logging, frame analysis, HyperFilters for isolating traffic up to 10Gbit/s and intelligent streams that lets the users create search criterias for frame loss, rate, latency, jitter and other kinds of combinations of measurements to identify problems in the system.

3.2.2 Distributed

This section goes through the related work that are proprietary which has implemented a similar distributed solution for generating and analysing packets.

Codemint Raspberry Wall

Codemint is a company where a developer, Erik Wramner, put together a couple of Raspberry Pis to implement a platform for performance testing web servers with affordable and realistic test cases [16]. During JFokus 2014 Erik presented this work and seems not to be a commercial product. It is specifically designed to do performance tests for web servers and is limited in that way. However it raises some of the questions this thesis aim to answer.

3.3 Summary

Demonstrably there are a lot of different software and products for generating and analysing network traffic. However these are all either too expensive or have a lack in functionality for what is needed for this thesis work. Many thoughts and ideas can be reused but no solution offers a cheap and also distributed system that theoretically can generate and analyse traffic indefinitely.

4
Design

In this chapter we present the design of the solution. We look at the problem at hand and then look at what parts we need and draw up a design from that. We go into detail of each part and how they interact with each other. We also go through what requirements that each part needs to fulfill and how these requirements are seen to in the design. The chapter consists of multiple sections where we in each section go through the details of each part of the system.

4.1 Design requirements

We start by looking at what we need the system to do. In cooperation with the supervisor at Layer 10 we establish a set of design requirements. With the help of these requirements we get an idea of what it is we want to achieve, and we can then evaluate the solution and compare it to other traffic testers. This section summarises these requirements.

Stream configuration

- The system should be able to send traffic up to a rate of 1 Gbps.
- The system should be able to configure traffic streams, one or multiple, that consists of custom configured network traffic. Different streams should be able to send different kinds of packets.

Packet configuration

- It should be possible to configure all fields of the relevant layers of the TCP/IP stack. The relevant layers are the Data Link, Network and Transport layer.
- The packet size should be configurable.

Traffic capture/counting

- The system should be able to count all sent packets with a 0% drop rate at rates up to 1 Gbps.
- The system should be able to inform of what the current incoming rate is, in a unit of bits per second or frames per second.
- The system should be able to capture packets for deeper inspection, not necessarily at the highest rates as this amounts to very heavy work loads.

- The system should be able to define filters that count only packets that match the filters.

4.2 System overview

With the requirements presented in the previous section we need to design a solution that gives us the tools we need in order to achieve this. In this section we present an overview of what the system should contain in order to emulate a traffic generator. For this we use a distributed system of nodes that are, in a synchronised manner, sending and receiving network traffic through a device that is under test. We need to be able to control and communicate with these systems without disturbing the traffic flow that is used for testing the device.

We look at what parts the distributed system consists of and how those parts interact with each other to achieve a solution to the problem. To our help we have Figure 4.1 that depicts the overview of the design. As can be noted in the picture the system consists of the following parts:

- Two different traffic flows - one for test traffic and one for control messages.
- One master node for control and communication.
- Two groups of nodes. One side for transmitting traffic and one side for receiving traffic.
- The device under test. Not really a part of the designed solution. However it is included for clarity.

Using this picture and bullet list we will present each part shortly and then in the following sections go through the details of each part. The first thing to notice in Figure 4.1 is that we have two flows of traffic. One flow of traffic where all control and communication messages are being sent and received between the nodes. Then one other flow where the actual test traffic flows through, this flow is separated from the control and communication network. The next thing to notice are the different groups of nodes. We have one master node that handles the communication and synchronisation of the system. Then there is one side of nodes that transmits the test traffic through the system and one side that receives all traffic and reports the results back to the master node. Lastly we have the device under test. It is not an actual part of the test system but showing where it is situated helps in giving clarity to how the solution is supposed to work.

In summary the system is made of two sets of an arbitrary number of nodes where one side is transmitting test traffic and the other is receiving test traffic. These are controlled by a master node that communicates with each side through a TCP/IP network, separated from the test traffic network. The test traffic is sent through the nodes that are connected to the device under test through another TCP/IP network.

4.3 The two traffic flows

The system has two different traffic flows, one for test traffic and one for communication. To ensure a stable stream of test traffic we want to separate these two traffic

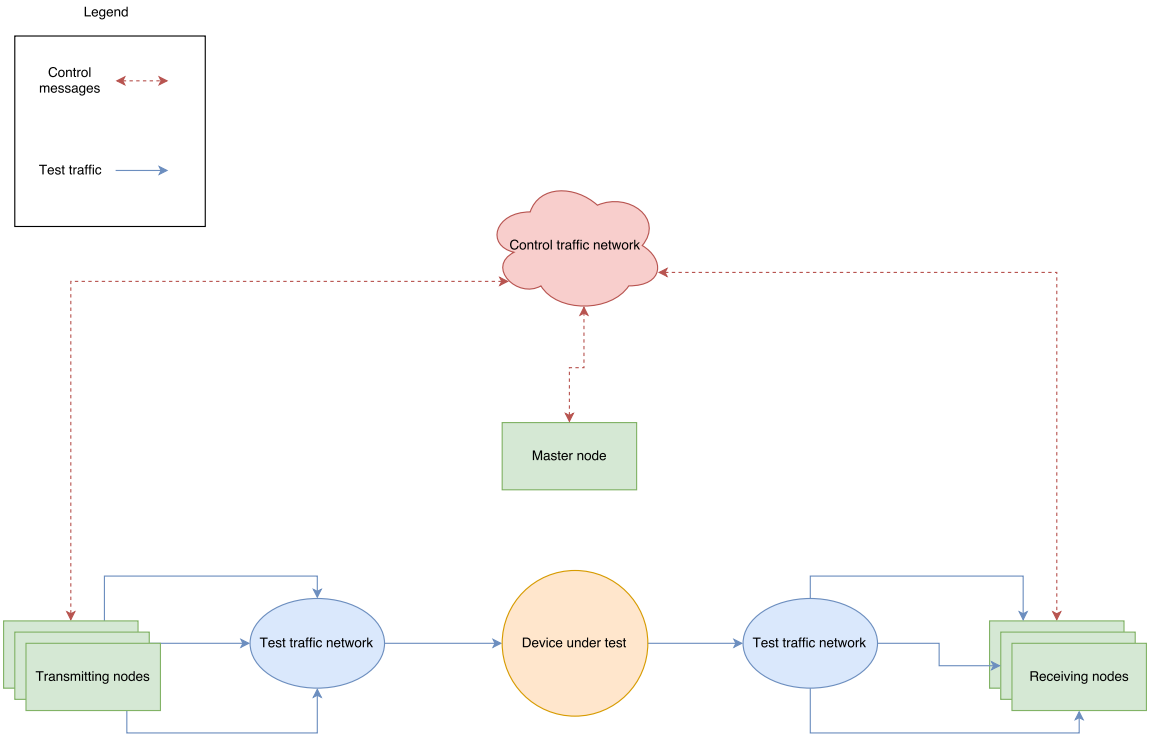

Figure 4.1: An overview of the system design.

flows. In order to separate test traffic from control and communication messages between the nodes we want to design the system to have separate networks. By separating the traffic flows we avoid problems such as filtering out communication messages from the test traffic so they are not mistaken for actual test traffic. If they are not even being received on the same interface, we do not need to worry about them mistakenly being treated as test traffic.

In this section we go through how these networks are designed and what requirements they have. Much like how the networks are split up we split this section into two subsections where each subsection looks at one network each. Subsection 4.3.1 presents the details of how the control network should work. In subsection 4.3.2 we look at how the traffic network is set up and how the traffic should flow through the system.

4.3.1 Control and communication network

To enable control and communication between the master node and the groups of transmitting and receiving nodes we need a network that enables communication over TCP/IP protocols. For this we need a set of requirements:

- Reliable communication between the nodes.
- Every node has a unique address, enabling communication.
- Separate control traffic from the test traffic, to avoid control traffic mistakenly being accounted for as test traffic.
- Communication has two flows. One flow between the master node and the transmitting side and one flow between the master node and the receiving

side.

These requirements put together are what make up the means of communication between the nodes. This network only contains the participating SBCs and not the device under test. In summary the network should enable communication in both directions between master node and each side respectively. Communication between transmitting and receiving side is not necessary. The master node initiates and handle all communication. Figure 4.2 depicts how the communication flows within the control network.

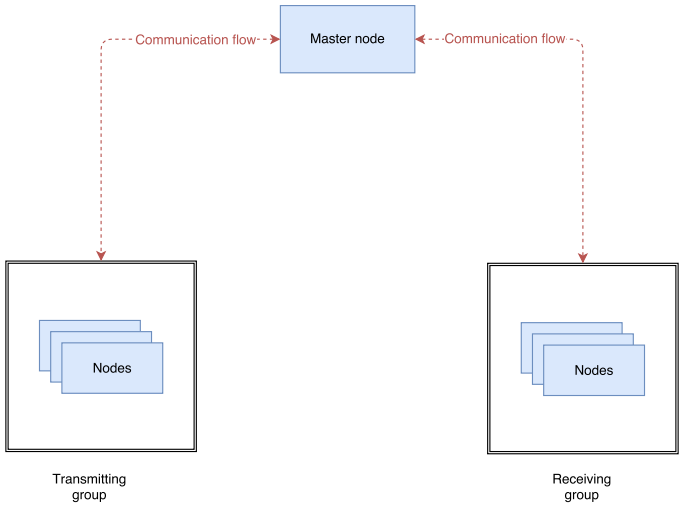

Figure 4.2: An overview of how the control traffic and communication flow between master node and the groups of transmitting and receiving nodes.

4.3.2 Traffic network

The traffic network is the network that contains the device under test, the switches and the transmitting and receiving nodes. This network should only handle the traffic flow which is used to test the device under test. Much like the communication network we set up a few requirements for the traffic network:

- The network needs to be able to reliably transfer high-speed Ethernet traffic without dropping packets on the medium.
- The network should be isolated to the test traffic so that no other traffic disturbs the testing.
- Each node in the traffic network needs to have a unique address so that the traffic can be routed correctly from one transmitting node to one receiving node.

The test traffic flows through the network from the transmitting nodes through the network into the device under test and is then forwarded by the device under test and finally arrives at the receiving nodes. This flow can be viewed in Figure 4.1.

4.4 The master node

In order to achieve a distributed system in which we have a group of nodes that transmit test traffic and a group that receives that traffic we need to be able to control when the nodes shall be transmitting and receiving. We need to synchronise these groups within and towards each other so that we are sure that when the transmitting side starts sending traffic all the participating nodes start sending at the same time. We also need to make sure that the receiving side is ready to receive the traffic when it starts flowing through the system. In addition to just transmitting and receiving we also need to distribute these tasks to the groups. One solution to this problem is to have a designated handler. A controller that can communicate with each group separately and synchronise them respectively. This is where the notion of a master node is introduced.

To control and monitor the system we need a master node that is dedicated to this task. The master node handles the communication with the groups of transmitting and receiving nodes. It is the master node that distributes the tasks, starts and stops the test traffic and collects the results. The master node's tasks can be summed up in the following list.

- Create the corresponding transmitting and receiving groups of nodes.
- Distribute the tasks of transmitting and receiving traffic respectively.
- Synchronise the groups.
- Collect the results of the sent traffic.

We can split the design of the master node into two components. One software component and one hardware component. The software component is the program that executes the master node's tasks. The hardware component is what the master node requires hardware-wise to be able to execute its program. In the following two subsections we will look at each component separately.

4.4.1 Software component

In the previous section we established the concept of a master node and why it is needed. However we have not yet stated how it should function, only that it is required. The master node is comprised of two components, one software and one hardware. The software component is the instructions that execute on the master node to allow it to fulfill its tasks. In this subsection we look at the design of the software component and how it functions. This subsection is split into four smaller sections where each section corresponds to one of the master node's tasks. Each such section go into detail of how the software enables the master node to execute its tasks.

4.4.1.1 Grouping into transmitting and receiving nodes

To be able to send the correct tasks to the correct nodes the master node needs to be able to divide the nodes into groups according to their roles. To know each node and contact them the master node also needs to be able to reach them through an address. According to one of the previous design requirements the control network needs to provide a unique address to each participating node, so we can use these

addresses for grouping. Given that the master node knows each address of the participating nodes in the control network it is just a matter of dividing these addresses into respective transmitting and receiving groups.

Dividing these addresses, or nodes, into the groups needs to be done equally. For every node in the transmitting group we need a node in the receiving group. Meaning that for n number of traffic flows we need $2 * n$ number of nodes. Looking at Figure 4.3 we can see how the software should split the addresses.

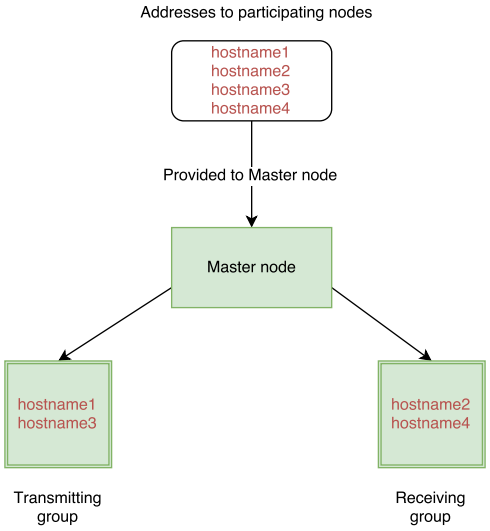

Figure 4.3: An overview of how the master node splits the provided node addresses into each transmitting and receiving group respectively.

When this is done, the master node has created two groups of nodes using their addresses. One is dedicated to sending traffic and the other group is dedicated to receiving it. The master node can use these two communication groups for controlling and communicating with each node knowing which group it belongs to.

4.4.1.2 Distributing the tasks of sending and receiving traffic

We have two different groups of nodes, one that should transmit test traffic and one that should receive. However, no one has told these nodes that they should be doing this yet. They are not aware of their current roles. As the master node is the handler of the system it needs to spawn and distribute each groups' task so that the participating nodes can carry out what they are intended to do.

So the tasks of sending and receiving need to be two well-defined processes, or software components, P_t and P_r where P_t is the process that is sending traffic and P_r is the process that is receiving traffic. They need to have well-defined starts and finishes and it is also necessary that the nodes still can communicate with the master node as these tasks are executed. We will go through these software components in detail in Section 4.5.

Given that the master node can acquire these two software components the task of distributing them is simple. The master node communicates with each group. It tells every participating node in the transmitting group to run process P_t and it tells every node in in the receiving group to run process P_r.

4.4.1.3 Synchronising the groups

We want to design a system that can send traffic from one group of nodes to another group of nodes that receives that traffic. In order to ensure a stable traffic stream and no packet losses we need to think about synchronisation. Not only stability but also ensuring that the contents of a packet stream at time t does not deviate when the same packet stream is started at another time t'. There are two scenarios that result in packet losses if we ignore synchronising the groups.

- Traffic loss if a transmitting node starts sending traffic before its corresponding receiving node has not yet started its receiving task.
- Traffic loss if the receiving nodes stop its receiving task before the corresponding transmitting node has finished sending traffic.

To achieve synchronisation between the nodes a construct called barrier synchronisation is relied upon. Barrier synchronisation is a simple concept. As the name implies a barrier is set up that does not let any process execute until all participating processes has reached this barrier. We can use the master node as a handler to accomplish this. The problem we need to solve is to make sure that no node in the transmitting group starts before the receiving nodes have started their receiving tasks. Furthermore we also need to make sure that the receiving nodes do not stop their receiving task before the transmitting nodes are finished with sending traffic. To accomplish this we can use barriers at two levels. The top level where barrier synchronisation happens between the master node and either receiving or transmitting group. The bottom level is where the barrier synchronisation happens between the nodes of a group, making sure they all start at the same time. There are in total four barriers in the system, three located at the top level and one located at the bottom level. Explaining these barriers is easiest done by going through how they are started in relation to each other.

1. The first barrier is done at the top level between the root node and the receiving group, it is there to make sure that each node is ready to receive traffic.
2. The second barrier is a bottom level one which is done between all transmitting nodes, making sure they start at the same time.
3. The third barrier is another top level barrier that is done between the root node and the transmitting nodes.
4. The final barrier is a top level barrier between the root node and the receiving nodes. It is started by the receiving nodes after they have reached the first barrier and is only finished when the root node has reached and passed the third barrier.

To further illustrate how this should work we introduce Figure 4.4 that depicts a flow chart of how the master node synchronises the group towards each other.

In Figure 4.4 we can observe that the synchronisation starts with the master node synchronising the group of receiving nodes to start their receiving tasks. When all receiving nodes are finished with starting their receiving tasks, the master node can go ahead and do the same with the transmitting nodes. Through this synchronisation between the master node, the receiving nodes and the transmitting nodes we can ensure a stable traffic flow without any lost traffic due to any node starting or finishing before they should have.

Then when the transmission should end the process is reversed. The transmit-

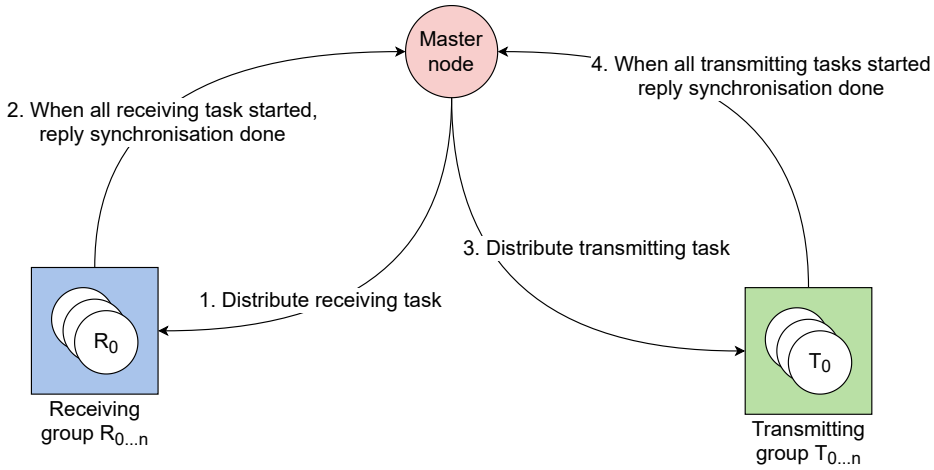

Figure 4.4: Process of synchronising the start of a transmission.

ting nodes all synchronise the stopping of the traffic and tells the master node when it is done. The master node then tells the receiving nodes to stop their receiving task. First when the receiving tasks are done and stopped their receiving tasks in a synchronised manner, the results can be extracted. This flow can be viewed in Figure 4.5

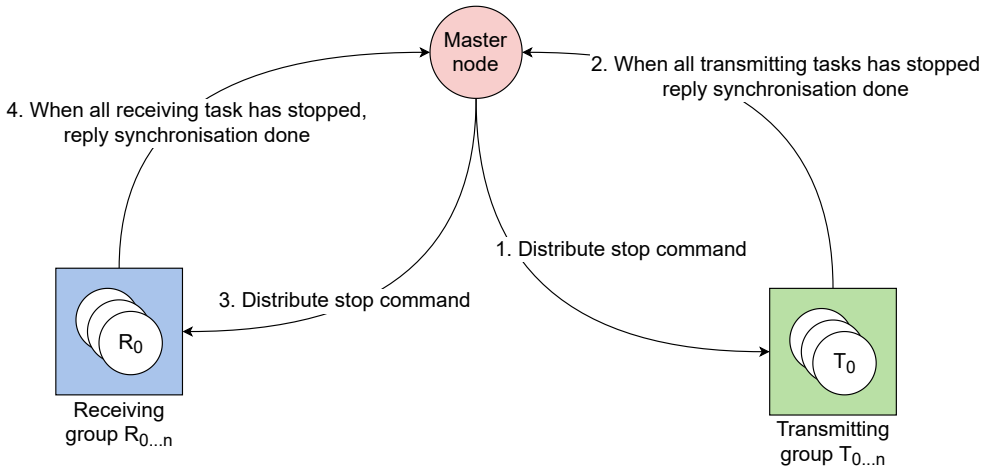

Figure 4.5: Process of synchronising the end of a transmission.

4.4.1.4 Collect the results of the sent traffic

When the traffic scenario has been concluded and all the participating nodes have, in a synchronised manner, finished their respective tasks and reported back to the master node, we want to be able to fetch the results of the captured traffic.

Since the capturing task is split up evenly between the receiving nodes, and each node only receives at most one packet stream, we need a way of obtaining each node's result back to the master node. The master node is how we interact with the system so this is where we would like to present the result of a traffic scenario.

If we can obtain each node's captured result, let us call the result of node x Cap_x, we can aggregate all receiving nodes' results to get a complete view of the captured traffic result. We can call that Cap_{tot}.

In addition to the packet capturing results we also need to obtain the information of how much traffic that was sent from the transmitting nodes. For transmitting node y we can call it $Trans_y$. In the same way we get Cap_{tot} we should be able to aggregate all the transmitted data and get $Trans_{tot}$. This is required for us in order to determine that no packets were lost during the traffic scenario. In other words $Trans_{tot} = Cap_{tot}$. We can then formulate following requirements on the master's node collection of captured traffic:

- We need a way to obtain each receiving node's captured traffic, $Cap_{0...n}$, and aggregate all of them to Cap_{tot}.
- We need a way to obtain each transmitting node's sent traffic, $Trans_{0...n}$, and aggregate all of it to $Trans_{tot}$.
- When the system has finished sending traffic $Trans_{tot}$ should be equal to Cap_{tot} i.e. $Trans_{tot} = Cap_{tot}$
- This obtained data need to be presented in some form to the user of the system.

4.4.2 Hardware component

The hardware requirements on the master node are not as strict as for the receiving and transmitting nodes. This is simply because of the fact that the master node is not sending or receiving any traffic. It is only handling the communication and control of the system. Thus we can limit the hardware requirements to following list:

- Standard network interface, either wired or wireless, to communicate over TCP/IP protocols.
- Processor with chipset able to run Linux.
- Some sort of operating system where so that the user can interact with the system.
- A persistant storage, for example a hard drive, where results can be stored, collected and inspected.

4.5 The transmitting nodes

The transmitting side of the system is responsible for outputting the network traffic out into the network and the device under test. Each node in the transmitting group is given a defined transmitting task. The transmitting task is interpreted, and the transmitting node should then be able to start and stop the traffic on command and then somehow save the result of how much traffic that was sent. The transmitting nodes should synchronise before traffic is started, and then again when traffic is ended. We can sum up the tasks of the transmitting nodes in the following list:

- Given a transmission task, interpret it and execute it.
- Synchronise start and stop of transmission.
- Save the result of the sent data.

Again we can split up the design of the transmitting side into a software and hardware component.

4.5.1 Software component

The software component for the transmitting group is not as complex as for the master node. The configuration for the transmission needs to be distributed within the group. Where each node can execute the task individually, and on command start and stop it. In this section we will look at the software component in detail

The transmission task should be a predefined set of instructions that each transmitting node, node x is notated as T_x, can then execute to start generating network traffic. The configuration of each task, i.e the specifics of what the network traffic should look like and which receiving node should have it, should be spread evenly among the transmitting nodes. Each node should have at most one traffic stream and one receiver, receiving node y is notated as R_y. When the configuration has been distributed the transmitting nodes should synchronise the start of the traffic. After this when either a stop command has been received, or the configured traffic flow has ended, the nodes shall stop the transmission in a synchronised manner and save the statistics of how much traffic was sent. We can describe the flow by using a scenario:

1. Given a total of four nodes, two transmitting (T_0, T_1) and two receiving (R_0, R_1).
2. Given one transmission task P.
3. The transmission task is then distributed to the transmitting nodes. T_0 receives task P denoted P_{T0} and T_1 as P_{T1}
4. When P has been distributed, each node has to synchronise within the group to each other and first after this the traffic can be started.
5. When all transmitting nodes, T_0 and T_1, have started their task P_{T0} and P_{T1} the transmission of the network has started. The traffic flow should be as following $P_{T0} \rightarrow R_0$ and $P_{T1} \rightarrow R_1$
6. When either a stop command has been received or the requirements of the traffic configuration has been met, nodes T_0 and T_1 synchronise towards the master node to inform it that traffic is done transmitting.
7. After synchronisation has been established, each node save the statistics of the transmitted traffic for later inspection.

4.5.2 Hardware component

The hardware component of the transmitting nodes are a bit more strict than for the master node. The reason is because these nodes will actually be working hard to send as much traffic as they possibly can. We can summarise the requirements of the transmitting nodes in the following list:

- A network interface for the controlling traffic, i.e. the communication with the master node.
- A separate network interface for the transmitting network traffic. Preferably a high-speed Gigabit Ethernet one.
- A processor with chipset capable of running Linux.

4.6 The receiving nodes

The receiving side is responsible for capturing and counting all the network traffic that the transmitting side has sent. You could see it as the antithesis of the transmitting side. Much like each node in the transmitting side, every node on the receiving side is given a predefined receiving task. The task should contain all the necessary configuration for a single node to start capturing traffic on its assigned network interface, the interface that is coupled with the transmitting side. There are more similarities in that when a task is given to a node the task is interpreted and the receiving group synchronises among each node and then tells the master node they are done. The capturing is stopped only when the master node tells the receiving group that it should stop. The results are then saved somehow, either on disk or messaged back to the master node for later inspection. We can summarise the tasks of the receiving nodes in the following list:

- Given a receiving task, interpret it and execute it.
- Synchronise start of capturing, stop only when master tells the group it is safe to do so.
- Save the result of the captured data.

Much like the previous sections we can split up the design of the receiving side into a software component and a hardware component.

4.6.1 Software component

In this section we will look at the software component of the receiving side in detail. The software component of the receiving side consists of a receiving task that each receiving node should execute individually, after the task has been distributed within the receiving group. When the master node tells the receiving group to stop capturing, only then should the nodes stop their capture and save the result.

The receiving task should be a predefined set of instructions, including configuration of what sort of network traffic that should be captured. When this task, task R has been distributed to a receiving node y, notated as R_y, node y should then synchronise within the receiving group and then start executing that task. After this when a stop command has been received from the master node, and not before, the capturing should be stopped and the results be somehow persisted for later inspection. We can again describe this by using a scenario:

1. Given a total of four nodes, two transmitting (T_0, T_1) and two receiving (R_0, R_1).
2. Given one capturing task task C.
3. The capturing task is distributed to the receiving nodes. R_0 receives task C denoted C_{R0} and R_1 as C_{R1}
4. When C has been distributed, each receiving node has to synchronise towards the master node, telling that they are ready for capturing traffic.
5. When the capturing process has started, the capturing nodes goes into a new barrier synchronisation towards the master node.
6. First when the traffic scenario is done and the network traffic has been stopped, the master node will synchronise toward the receiving group, telling them it is fine to close down the capture.

7. After synchronisations has been established, each node saves the captured result somehow for later inspection.

4.6.2 Hardware component

The hardware component of the receiving nodes are very similar to the transmitting nodes. The receiving group need to be as fast as the transmitting nodes or even faster. We can explain this by the fact that the receiving nodes are always reacting to new packets coming in. The transmitting nodes are sending predefined network packets. We can summarise the requirements of the receiving nodes in the following list:

- A network interface for the controlling traffic, i.e. the communication with the master node.
- A separate network interface for the receiving network traffic. Preferably a high-speed Gigabit Ethernet one.
- A processor with chipset capable of running Linux.

4.7 Packet generation - the transmitting task

In this section we look at what the transmitting task, that gets distributed between the transmitting nodes, looks like. In the previous section we have established that in order to reach higher packet rates for packet generation the task of transmitting packets must be distributed between the nodes. The point of the system model is to share the burden of packet generation between the distributed nodes and combine the performance in order to achieve better rates of packet generation. To distribute a task the task must have a clear definition. For this system it makes sense to define the task as a configuration of a packet stream that each node can then start sending. The configurations defines the specifics of how the stream will send the packets. That is which rate each individual node will send in, if there are any delays between each sent packet or if the stream should send in bursts. To have a stream of packets you also need the actual packets. So in addition to a packet stream configuration, we also need a configuration for what the packets in that stream should look like. The contents of the packets should ideally be defined by the user. So the distributed task can be defined as:

- A packet configuration, what the packets will consist of.
- A stream configuration, how the packets will be sent.

The configuration is done by providing the master node with the configuration details for the packet contents and the stream. When the configuration is done it is distributed by the master node to the participating nodes. Each participating node replies to the master node's barrier synchronisation when the configuration is finished, and the master node keeps track of each answer and does not continue executing until every node has replied. When every node has replied the master node continues with distributing the capturing task among the receiving nodes, and ensures in a similar way that the system does not continue until each node is done with the configuration. More about the packet capturing in Section 4.8. When the capturing nodes are ready, the transmission can start. However it is important

that the transmitting nodes start at the same time. Otherwise there might be inconsistencies between tests as the aggregated stream will vary in size if not all nodes transmit at the same time.

When every node is synchronised, each transmitting node will start its transmission to its assigned receiving node. The switch aggregates the traffic and the total output from the switch to the device-under-test should be equal to the sum of the individual capacities of every participating node. Equation 4.1 presents a simple mathematical model to express the total output from the system.

$$P_{tot} = \left(\sum_{i=1}^{n} p_i \right) * k \tag{4.1}$$

Where:

- P_{tot}: is the aggregated packet rate of the system.
- n: is equal to the amount of participating nodes.
- p_i: is the individual packet rate of node i.
- $k : 0 \leq k \leq 1$: is a degradation factor.

Hopefully k should be equal to 1, however it is possible when the amount of nodes reach a high enough number this factor might play a bigger role. For example the switch might not have enough performance to handle every node's packet stream when the number of nodes increases and thus the resulting output will be limited to the switch's performance. If the switch only has a capacity of 80% compared to the aggregated performance of the transmitting nodes then $k = 0,8$.

4.8 Packet capturing - the receiving task

In this section we present what the receiving task, the task that is distributed between the receiving nodes, should consist of. By distributing the task between the nodes we can share the load and achieve higher rates of packet capturing. To be able to distribute the packet capturing it needs to be defined as a clear and consistent task, so that each node will execute the same thing. There are several ways to define a packet capturing but for this system it makes sense to limit it to three different items. A packet capturing task is defined by:

- The packet filter, whether the node should filter on specific packets.
- Counting or capturing mode, whether the packets should be discarded after arrival or if it should somehow be persisted for later inspection.
- Which interface that should be used for capturing. This should be the corresponding Ethernet interface.

The configuration is done in the same way as for the packet generation. User connects to the master node and provides it with the necessary configuration according to the three previous points. The task is then distributed by the master node between the receiving nodes and after synchronisation has been achieved, each node executes its receiving task.

4.9 Summary

In this chapter we have presented a suggestion to a solution for this system. We started by looking at what minimum requirements we had for the system to be viable. We then presented a system overview of what components the system should include in order to function as a network tester. We then started looking at each component in detail. Beginning with the two traffic flows, or networks, and what their functions are and why we need to split the network into two. In the section following this we presented the master node and its tasks, how it communicates and synchronises with the receiving and transmitting nodes and what sort of minimum hardware requirements it had. We then looked at the transmitting group and the receiving group, their tasks and how they function in relation to each other. Finally we looked into the two different tasks that the transmitting and the receiving group shall execute. We defined these two tasks of what they should do and how the groups should execute them.

5
Implementation

This chapter aims to present the details regarding the actual implementation that resulted in the prototype for this system. In this chapter we will go through each part of the system in detail. The chapter will be split up much like Chapter 4, so that we can easily connect the actual implementation to the suggested solution. The chapter is split into three bigger areas. In the first part we will look at a system overview of the developed solution and also present the operation of the system in order to get a bird's eye view of how the system works. In the second part we will look at the actual implementation of the different parts of the system and go into detail of each one of them. In the last part we will look at the hardware that was chosen for the system. The reason to why its all presented by itself is because the transmitting group, the receiving group and the master node all ended up using the same hardware.

5.1 System overview

The resulting solution consists of a total of seven single-board computers of the brand ODROID-C1, one Ericsson Ethernet switch, a wireless router and a lot of RJ45 cables. Out of the seven SBC, three boards are assigned to the receiving group and three boards are assigned to the transmitting group. The final SBC acted as the master node, that handled the interaction with the user and the overall operation of how the system transmitted and received data.

The transmitting group generates packets where each node transmits to at most one receiving node each. The traffic goes through the Ericsson switch and is delivered to each receiving node by MAC address.

5.2 Operation

To control the system the user connects to the master node, either directly via a monitor and keyboard or remotely using e.g. SSH. The master node contains a set of instructions that is used for configuring the specifics for transmitting and receiving. When configuring a test scenario the user needs to supply the system with an instruction that specifies which nodes that should be used in the test. When configuration is done the master node distributes the configuration among the partaking nodes and wait for the system to synchronise. When each node has synchronised the master node will start the test. During the transmission the user can also send commands to the master node for status information, such as what

the current packet rate is. The master node then prompts each receiving node of what its current packet rate is and then aggregates each reported packet rate and presents it to the user. When the transmitting nodes have finished transmitting they inform the master node that the transmission is done. The master node then stops the receiving nodes, collects the results and presents it to the user. Figure 5.1 displays a sequence diagram of how a test is typically done.

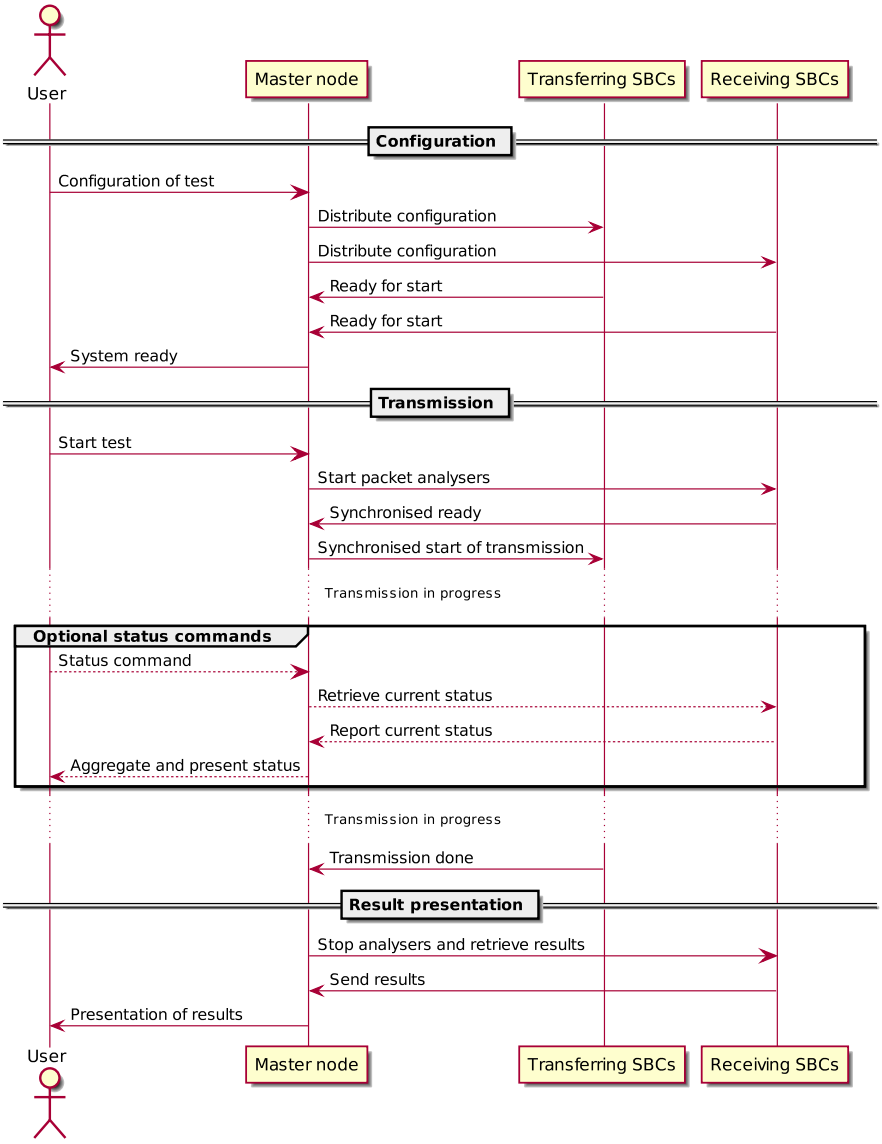

Figure 5.1: Sequence diagram of a typical system execution.

5.3 Master node

The implementation of the master node is built in Python and uses a framework called `mpi4py` that is a framework for using MPI calls within Python. This also requires a working MPI implementation, we use an implementation called MPICH [6]. We also considered using `dispy` [17], but it was harder to use and implement

towards. For the hardware we used the single-board computer ODROID-C1, the specifics of the ODROID can be read about in Section 5.6. Before we go into the implementation of the master node we recap the main tasks of the master node in the following list.

- Create the corresponding transmitting and receiving groups of nodes.
- Distribute the tasks of transmitting and receiving traffic respectively.
- Synchronise the groups.
- Collect the results of the sent traffic.

We want the software of the master node to create the communication group, then distribute the respective task to each group, synchronise them and then collect the results. In Appendix A we present the software implementation of the master node in Listing A.1. In Figure 5.2 we present the software implementation execution from the master node's point of view.

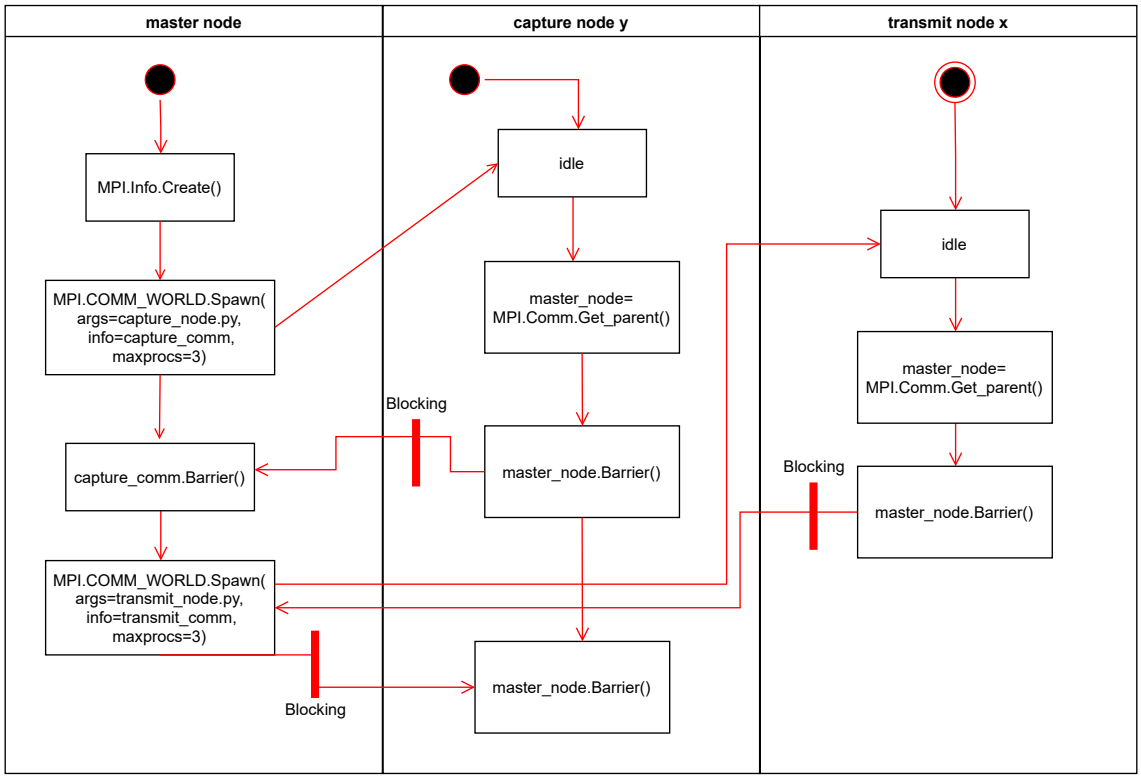

Figure 5.2: Software execution flow from master node's point of view

The master node starts with fetching the addresses for the receiving nodes by reading the predefined capturing host file. It uses the call `MPI.Info.Create()` to create a key-value pair that can be passed around to different messaging groups. It uses this key-value pair to store the predefined addresses for the capturing nodes. It should be stored in a file called `caphosts` in the directory `hosts`. It then continues with creating the communicator object for the capturing group. This is how we communicate with the receiving nodes. So the call `MPI.COMM_WORLD.Spawn()` spawns our receiving group and uses the `capture_info` to figure out which hosts to spawn the process of the `capture_args`. It spawns at most 3 processes, which is the number of the receiving nodes, according to `maxprocs=3`.

We then reach the first barrier. And as can be viewed in A.1 it is made on the communicator to the receiving group. This ensures that the master node's program does not continue to execute until all of the children nodes in the communicator have reached the respective barrier in their program. When the receiving nodes are finished synchronising we continue with setting up the transmitting group's communicator much in the same way that we did for the receiving group. It is identical, with the difference that we use another host file that instead lists the addresses of the nodes that should be transmitting traffic rather than receiving it. After this we have a synchronisation on the transmitting nodes that are the final synchronisation the transmitting group does when they have finished transmitting. The master node will wait here until all transmitting nodes has reached this point. When they are all finished we continue with synchronising towards the receiving group as now we know that they can stop capturing traffic as well. After this, the master node's program is finished and we can extract the result from the receiving group.

In summary with the help of the master node's program we have created the transmitting and receiving group of nodes. We then distribute the tasks of transmitting and receiving task respectively. Throughout the program we synchronise the groups at critical moments in order to ensure traffic consistency. Finally we do not collect the results from the master node. The reasoning behind this decision is that the project ran out of time and this feature was not prioritised as the results could just as easily be extracted from the receiving nodes manually.

5.4 Receiving nodes

The receiving nodes' capturing task are much like the master node built in Python, and uses mpi4py for the MPI calls and functions. On the nodes MPICH, the implementation of MPI, is also installed because it is what mpi4py uses under the hood to execute the calls. For the actual capturing of network traffic we use the software `netsniff-ng`. It is a highly performant packet sniffer that use zero-copy mechanisms so that the kernel does not need to copy packets from kernel space to user space and vice versa [18]. It is extra beneficial when you are using the limited hardware of a single-board computer. We use the same single-board computer ODROID-C1 for the receiving group as for the master node. Let us recap the tasks of the receiving group before we present the software implementation.

- Given a receiving task, interpret it and execute it.
- Synchronise start of capturing, stop only when master tells the group it is safe to do so.
- Save the result of the captured data.

In Appendix A Listing A.2 we present the software implementation of the receiving nodes. In addition to the listing we visualize the flow of a receiving node in Figure 5.3.

A member of the receiving group starts with establishing a connection to the master node with the use of the MPI call `MPI.Comm.Get_parent()` We then define the capturing task with a class wrapper for netsniff-ng called PacketCapturer. We set up the parameters to netsniff-ng by telling it should listen to interface eth0,

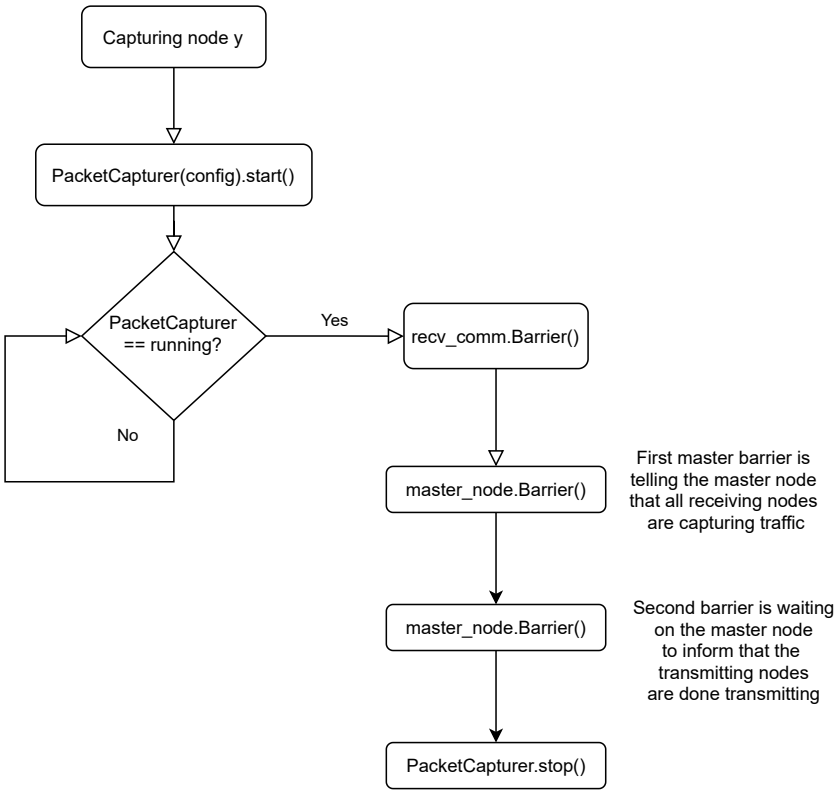

Figure 5.3: Flow chart of a receiving node.

run in silent mode, run in high-priority mode in order to achieve a higher scheduling rate and finally to set the RX ring size of 500 MiB. The incoming packets are stored in the RX ring until the device driver can process them. Which is why if we increase it we can reduce the amount of lost packets due to the RX ring being overfull. After the PacketCapturer has been configured we start the netsniff-ng process by issuing a start command. After this we enter a while-loop that breaks only when we have made sure that the netsniff-ng process is ready by capturing its command output. After this we put up a barrier synchronisation within the receiving group and only continue once every node has reached this barrier. Meaning that all the receiving nodes have started their respective capturing processes. When the packet capture process has started we enter the barrier synchronisation with the master node, and we do not continue executing until the parent also has reached this barrier. This happens only after the transmitting nodes have finished their sending and the master node has synchronised this. First after this the master node enters the barrier, ensuring that no traffic is lost due to the receiving nodes stopping their packet capture before the transmitting nodes have finished their sending. We finish by adding a small sleep timer so that the netsniff-ng process has enough time to finish any eventual writing to disk from the packet capturing.

In summary we have presented the software implementation for the receiving group. It ensures that a receiving task is distributed to each node and then is executed. We have shown how it handles the synchronisation both between the nodes of the receiving groups and towards the master node to ensure no packets are lost. The result of the packet capturing is saved to disk by netsniff-ng for later

inspection.

5.5 Transmitting nodes

The implementation of the transmitting nodes is also built in Python with the same framework of mpi4py to utilize MPI calls. The implementation for MPI, MPICH, is again installed, as it is this implementation that mpi4py is making its calls to. In addition the transmitting nodes also use the kernel module `pktgen` for sending the actual network traffic. The kernel module is included in the Linux kernel, however it is not enabled by default. For the transmitting nodes this is done when compiling the Linux kernel. When the module is enabled and running `pktgen` creates a thread for each CPU core, monitoring and controlling is done via the `proc` files. The strength of pktgen is that it bypasses most of the network stack and is thus a very efficient access to the NIC driver [19]. This helps us in squeezing more performance out of the limited hardware. The hardware is the same as for the master node, an ODROID-C1. More can be read in Section 5.6. Before we look into how the transmitting group is implemented we recap the tasks that was established in Section 4.5:

- Given a transmission task, interpret it and execute it.
- Synchronise start and stop of transmission.
- Save the result of the sent data.

With these requirements in mind we go through the algorithm for the transmitting nodes. The software implementation is presented in Appendix A in Listing A.3. To further visualize the flow of a transmitting node we present a flow chart in Figure 5.4.

The algorithm for a transmitting node starts with establishing a connection to the master node through the `MPI.Comm.Get_parent()` call. We also establish a connection to the communication group of transmitting nodes with `MPI.COMM_WORLD`. Each transmitting node receives a unique id, called rank, with the `comm.Get_rank()` call. We also save the number of nodes within the transmitting group in a variable called size. The rank is used to make only one node do the actual configuration for the packet transmitting, it is then this configuration that is spread to each participating node.

Each node gets a unique rank and there it is only one node that does the actual configuration. The node with the id, or rank, 0 is the one who sets up the configuration and saves it in a configuration array that is later distributed to each node. In the specific scenario presented in Listing A.3 the following configuration for the packet of stream is:

- A packet size of 64 bytes (`pkt_size`)
- Destination IP of 192.168.0.1 (`dst`)
- Destination MAC address of 00:1e:06:c2:11:13 (`dst_mac`)
- A total packet amount of 100 000 (`count`)
- Packet is stored in one sk_buffer that is cloned 100 000 times (`clone_skb`)
- A send rate of 50 000 packets per second (`ratep`)

The reason to why the configuration setting of `clone_skb` is used is because of increasing performance [19]. This setting offers the possibility to instead define the network packet once, in one sk_buffer, and then clone it. Rather than creating

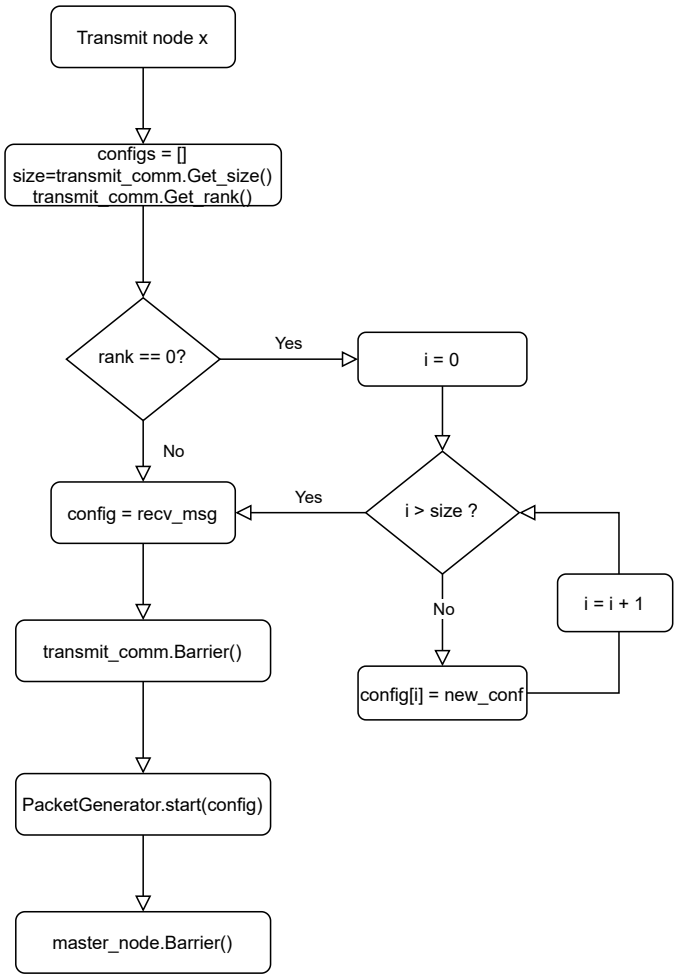

Figure 5.4: Flow chart of a transmit node.

the same packet several times. This is useful for this specific packet stream where we are sending an identical packet many times. We can increase the sending rate perfomance significantly by using this method. If we wanted a packet stream consisting of different packets, we would not be able to use this configuration setting. The configuration is then scattered with the MPI call `comm.scatter(configs, root=0)`.

We then use a class wrapper of pktgen called PacketGenerator to set the configuration. After this we set up a barrier synchronisation in the transmitting group, ensuring every node has finished configuring before we continue executing. When synchronisation has been achieved we start the transmission of the packet stream. This call is blocking until all the 100 000 packets have been sent. We then enter a new barrier synchronisation, this time with the master node. When every transmitting node has finished sending and reached this barrier the master node knows that the traffic scenario is over and it can safely tell the receiving nodes to stop capturing traffic. After the synchronisation with the master node is finished, the software is done executing and the connection is closed.

In summary we define a transmission configuration. The configuration is distributed to each participating node. We then synchronise each node toward each other within the group. The configuration is then executed by each node running

`pktgen` until the traffic is finished. We then synchronise towards the master node, making sure each node has finished sending traffic, and then we disconnect and the program is finished.

With all the details presented, in Figure 5.5 we present a simple relations diagram how all the Python scripts and classes relate to each other.

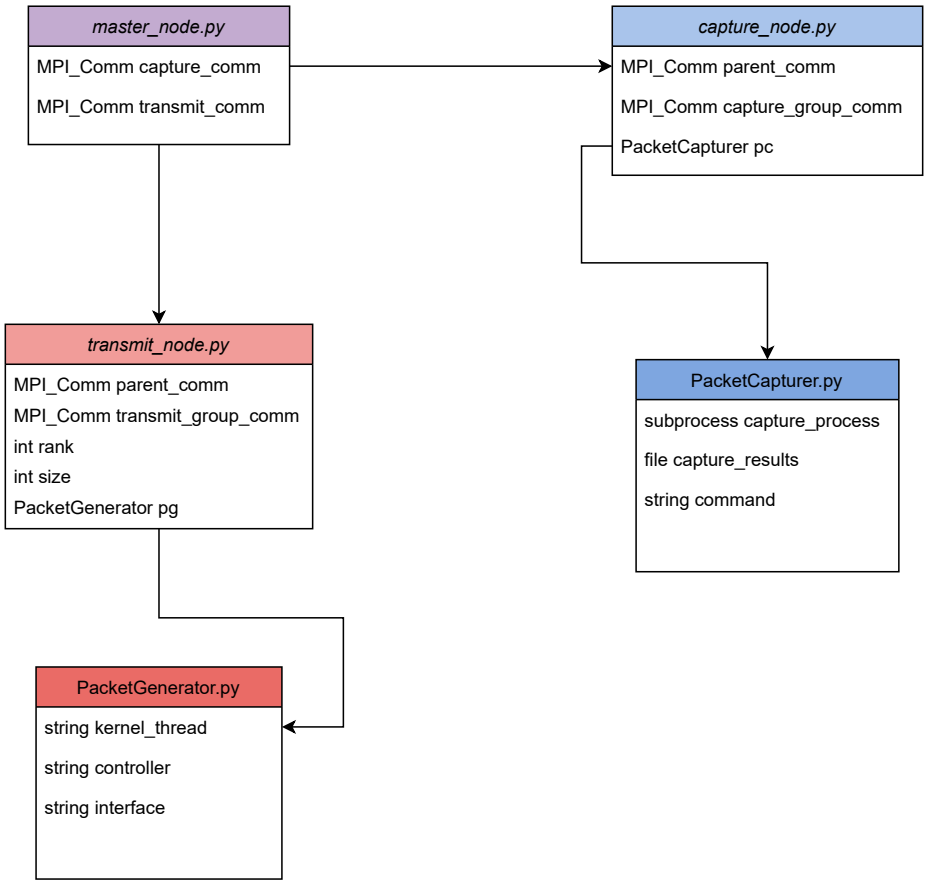

Figure 5.5: A relations diagram.

5.6 The hardware

In this section we present the hardware we use for the implemented solution. We look at the design requirements and then present the hardware that was chosen. In the first section we go through the hardware choice for the master node, receiving group and transmitting group, i.e. the single-board computers. They all use the same board even though the requirements for the different groups was not necessarily the same. However it made it easy for us in utilising the same board for all of them. After this we also present the switch that is used as an aggregating device into the device under test. The switch was not really a part of the thesis work, however still an integral part in the resulting system.

5.6.1 The single-board computer - ODROID-C1

When looking at what sort of hardware we wanted to use for the system we used the design requirements as guidelines. A small pre-study was made looking at different boards that is presented in detail in Section 5.6.3. Looking from a hardware perspective the requirement regarding being able to send in 1 Gbps packet rates was a key requirement. In order to make it easy for us and without the risk of any node not being able to handle the traffic stream we decided to primarily look at single-board computers that could handle 1 Gbps traffic rate. It was also important that the cost should be kept low, otherwise we would defeat the purpose of the system, which is to build a network traffic generator that is cheaper than the proprietary ones.

With the requirement of 1 Gbps in mind we looked at three key components when deciding on the candidate for a SBC. These were the processor, the network interface and the RAM. It is important to find a good quality, performant processor that can handle processing as many packets as possible. The reasoning being because out of all the single-board computers that were taken into account, none of them had network a network interface with any advanced features such as Direct Memory Access (DMA). DMA is good for network oriented systems because it helps in putting off load from the CPU by being able to access the memory directly and move data from the NIC to the memory. So when we realised we would not be able to find a single-board computer, at the time, with this feature we had to try and find a better CPU that can somewhat balance out the loss of such features. The RAM, and amount of RAM, is important to take into account because this is where the packets get stored temporarily waiting for the CPU to handle them, specifically in the RX/TX rings. So with a limited hardware where the CPU will for many cases not be fast enough we need to be able to buffer up network packets in the RAM memory, the larger the better.

In addition to the hard requirement on the speed of the network interface card we also had requirements regarding the possibility of having two separate network interfaces, one for the testing traffic and one for the communication traffic, and a kernel that can run Linux. With all these features in mind we decided on choosing the ODROID-C1. In Section 5.6.3 an evaluation is presented between the different platforms that were considered. The ODROID-C1's key features are the following:

- 1.5 GHz Amlogic S805 Quad Core Cortex A5 processor.
- Integrated Gigabit Ethernet transceiver called Realtek RTL8221F.
- Samsung K4B4G1646D : 1GB DDR3 32bit RAM.

The ODROID met all the requirements we needed. It also came with a WiFi adapter which enabled us to have two separate network interfaces. The integrated network interface did not have any DMA, multi queues or any other more advanced features which means that it is up to the CPU to handle all incoming and outgoing network traffic. For single-board computers the processor featured in the ODROID-C1 is luckily relatively performant. The 1 GB RAM also helps to buffer the packets the CPU does not have time to handle right away. When the work was carried out this SBC was one of top contesters for all-purpose single-board computers.

In summary we made sure to choose a single-board computer that was not the most expensive one there was to find. However we made sure to select one that still met all the requirements, even if it was cheaper than the more performant models.

5.6.2 The aggregating switch - Ericsson SP 210

The switch that ties the system together is one provided by Ericsson. The SP 210 is a packet aggregation node optimised for packet networks with routing and switching functionality at line rate of 36 Gbps, which is more than enough for the nodes used in this thesis work. It has 8 RJ45 ports and also additional 8 SFP ports if the system needs to be extended. It supports the Layer 2 protocols that were needed according to the design requirements, such as 802.1Q VLAN tagging. It guarantees line rate from 64 byte to 9600 byte MTU, which is necessary since the packet streams can consist of any MTU in that span.

5.6.3 Pre-study of the hardware

This section presents the small pre-study that was carried out in order to find which platform would fit the application best, i.e. which SBC that would make out the core of the system. The guidelines were simple in that it should not be too expensive and it should be easy to obtain. The two main factors in choosing a SBC were the network interface and the CPU that was going to be able to handle all the data that was coming from the network interface. In Table 5.1 the different boards that were considered can be seen and what properties of the boards that were the most relevant for the application.

Name	Processor	Memory	Network	Storage
Banana Pi	ARM Cortex-A7	1 GB DDR3	1 Gb Ethernet	SD, SATA
Banana Pro	ARM Cortex-A7	1 GB DDR3	1 Gb Ethernet, 602.11n	microSD, SATA
Raspberry Pi B+	BCM2835 ARM11	512 MB SDRAM	100 Mb Ethernet	microSD
BeagleBone Black	ARM Cortex-A8	512 MB DDR3	100 Mb Ethernet	Internal 2 GB, microSD
Miniand Hackberry	ARM Cortex-A8	512 MB DDR3	100 Mb Ethernet, 602.11n	Internal 4 GB
Cubieboard	ARM Cortex-A7	1 GB DDR3	100 Mb Ethernet	Internal 4 GB, microSD
ODROID-C1	Cortex-A5 ARMv7	1 GB DDR3	1 Gb Ethernet	microSD

Table 5.1: Comparison of different single-board computers.

The boards that are marked in bold are the ones that were chosen for testing if they could work in this system the system. Mainly because of the 1 Gigabit Ethernet interface, the cheap prices and the community's recommendations. The reason to why the Raspberry Pi was chosen as well, even though it only had a 100 Megabit Ethernet interface, is because the Raspberry Pi is very cheap and very popular so it is not hard to get a hold of. It was also interesting to have a Raspberry Pi for comparison, if it made a big difference to have access to 1 Gigabit Ethernet.

6
Evaluation

In this chapter we present the evaluation of the proposed solution. We split the chapter up in the different parts of the system. First we evaluate the packet generation of the system. We also present an evaluation of the hardware in a packet generating sense. In addition to this we also evaluate different software for the packet generation task. In the section after this we do the same with the packet receiving side. We then also discuss different versions of the packet generation task, and the pros and cons of it.

6.1 Packet generation

In this section we set up a test scenario and look at how well the system performs from a pure performance standpoint. We begin by setting up the test methodology, we continue with presenting the results. Finally we analyse the results and discuss benefits and drawbacks of the implementation.

6.1.1 Test methodology

The methodology is simple. We want to be able to find the hard limits of the system's transmitting capabilities. In order to do this we need to send traffic as fast as we can and see if can find the sweet spot where the degradation factor k presented in Section 4.7 starts playing a bigger role. To do this we use in total seven nodes. We set up three nodes as transmitting and we set up three nodes as receiving, and one node as master node. In between we use the actual proprietary tester, an Anritsu MP1590B, that can give real-time traffic statistics. How much traffic it is receiving, what sort of traffic that is currently being sent through it. The Anritsu will act as the device under test. Figure 6.1 displays what the setup looks like. We summarise the test steps in the following list:

1. One packet stream per node.
2. Run traffic for ten seconds.
3. Observe live packet rate through the Anritsu.

6.1.2 The test scenarios

For evaluating the system we have used a few different scenarios. By testing a few different sized packets at different traffic rates we are trying different extremities of the traffic characteristics. What we are hoping to observe is that the system reaches

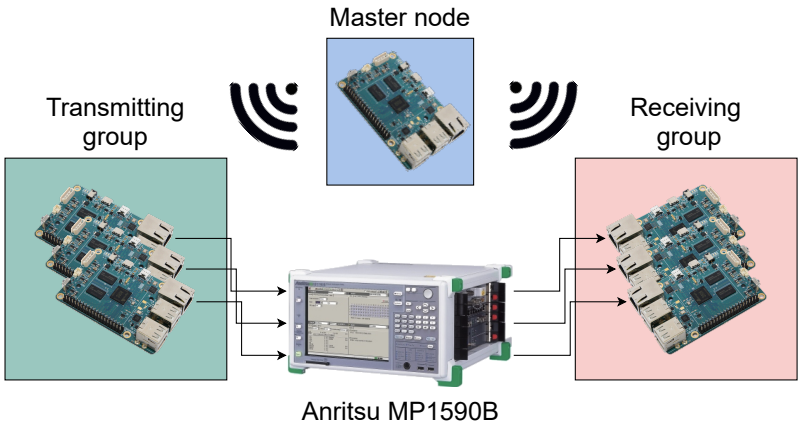

Figure 6.1: The test setup.

the given traffic rate within a plausible time frame, around one second. In order to pressure the system we chose traffic characteristics that we thought would be taxing for the CPU and the network interface card. We tried using a small, medium and large sized packet at different packet rates. We believe sending these packets as fast as we can to try and overwhelm the system, much like a denial-of-service attack, will help us in finding the bottlenecks. In Table 6.1 we present the different traffic scenarios we executed on the system. These scenarios are split over three different traffic rates. Where a traffic rate is how many packets are sent per second. In addition we test each traffic rate with three different packet sizes, going from smallest to largest. In total this gives us nine different test scenarios. What we want to observe is that the system behaves according to the simple mathematical model presented in Section 4.7. Which is that if we were to assign a traffic rate of 30 000 packets per second per node, then we should see, if we do not have any degradation, a total output of 90 000 packets per second, given that we are using three nodes for generating traffic.

Scenario	Size (bytes)	Send rate (pkts/s)
1	64	30 000
2	512	30 000
3	1500	30 000
4	64	60 000
5	512	60 000
6	1500	60 000
7	64	110 000
8	512	110 000
9	1500	110 000

Table 6.1: Different test scenarios

The results were extracted from the Anritsu after the test was finished. The results are split up into three graphs, one graph per packet rate. Each plot represent one packet size each, depicted in the legend. The graphs are plotted from the data extracted by the Anritsu where the Anritsu reports a packet rate every second. In

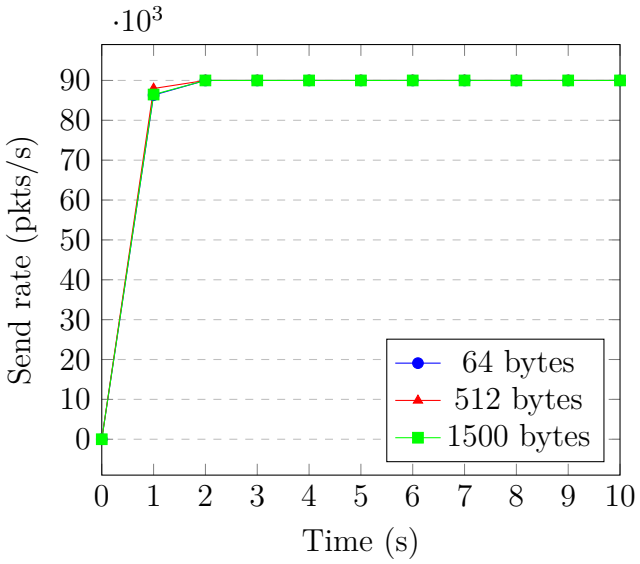

Figure 6.2: Plotted traffic results from scenario 1 to 3

Figure 6.2 traffic scenario one to three is presented where we used a traffic rate of 30 000 packets per second.

As we can see in the graph there is a small ramp up until the system reaches the designated packet rate of 30 000 packets per second per node. However once it is reached it is stable for the rest of the test. If we apply the mathematical model in Equation 6.1 that we originally presented in Section 4.7, we can observe promising results.

$$P_{tot} = \left(\sum_{i=1}^{n} p_i \right) * k \tag{6.1}$$

In Table 6.2 we can view the result of applying the mathematical model to the results of the tests. We find that the degrading factor k in the scenario when we have a packet rate of 30 000 packets per second is equal to 1. Meaning that there is no degrading factor of the system at all. The given packet rate provided by the user is what the system then performs.

Size	P_{tot}	n	p_i	k
64	90 000	3	30 000	1
512	90 000	3	30 000	1
1500	90 000	3	30 000	1

Table 6.2: Finding the k with 30 000 pkts/s

In Figure 6.3 we present the test scenario where we increased the packet rate to 60 000 per node. In Figure 6.3 we see the respective plots for the different packet sizes with a packet rate of 60 000 per node. Already at this packet rate we can observe a severe degradation in one of the scenarios. When we are using 1500 bytes for the packets we see we are already getting almost exactly half of the performance. The reported packet rates from the Anritsu flutter between rates of 97 000 to 99

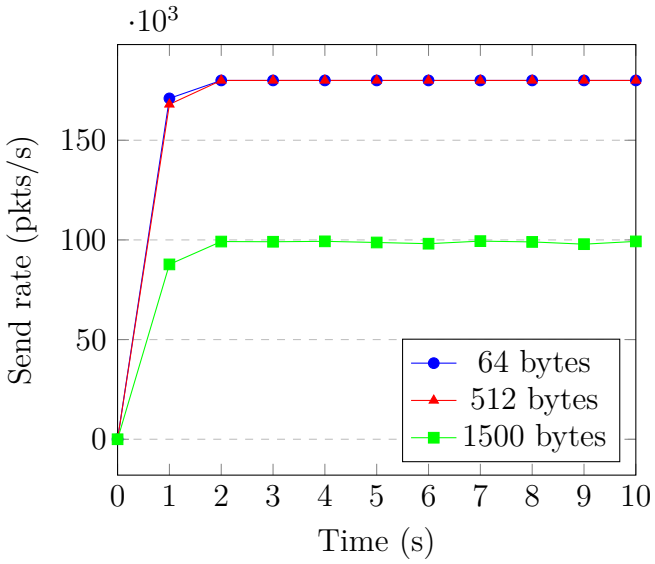

Figure 6.3: Plotted traffic results from scenario 4 to 6

000 packets per second. It seems to also be affecting all three nodes equally, in a stable way. We can look at the degradation factor in Table 6.3

Size	P_{tot}	n	p_i	k
64	180 000	3	60 000	1
512	180 000	3	60 000	1
1500	$\approx$ 98 000	3	60 000	0.54

Table 6.3: Finding the k with 60 000 pkts/s

By solving for k we can establish that we get a degradation factor of 0.54 when we are running with a packet rate of 60 000 packets per second and with a 1500 bytes packet size. The system is running only at a 54% efficiency, a huge degradation. To further investigate this we look at the last traffic scenario as well.

The final three traffic scenarios is presented in Figure 6.4. For this scenario we are trying to run the same scenarios as before but with a packet rate of 110 000 packets per second. Now we see a degradation even with packet size of 512 bytes, and the degradation is even worse for the traffic scenario with 1500 bytes. In Table 6.4 we present the calculations of the degradation factor k for the final three traffic scenarios.

Size	P_{tot}	n	p_i	k
64	330 000	3	110 000	1
512	$\approx$ 287 000	3	110 000	0.87
1500	$\approx$ 98 000	3	110 000	0.30

Table 6.4: Finding the k with 110 000 pkts/s

As can be noted the degradation factor now even seems to affect the lower packet sizes. What is interesting is that the packet rate has stayed the same for the traffic

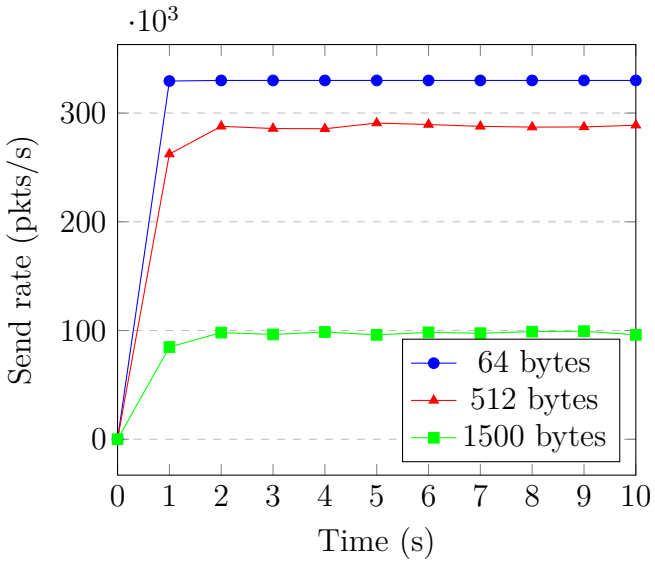

Figure 6.4: Plotted traffic results from scenario 7 to 9

Size	Packet rate (pps)	Throughput (Mbps)
64	110 000	56.32
512	≈ 96 000	393.216
1500	≈ 33 000	396

Table 6.5: Conversion to Mpbs

scenario featuring packet sizes of 1500 bytes. This suggests a hard limit somewhere in the system. If we convert the packets per second and packet sizes to another unit we can get a clue to where that limit is. If we calculate the throughput of each node to bytes per second instead by multiplying the packet size with the packet rate and then converting it to megabit per second we observe something interesting. In Table 6.5 we present these calculations for the last three traffic scenarios. Note that the throughput is given per node instead of the aggregate. These numbers suggest that we get a hard limit at pretty close to 400 Mbit, half of the ODROID's 1 Gbit.

To investigate this further and to eliminate the system as a potential source of error we performed a quick evaluation of all the different single-board computers' capabilities to generate traffic. The findings of our evaluation show that the ODROID-C1 has a flaw in that it only has 2 TX queues. It has 4 RX queues so it can receive traffic up to 1 Gbps without any issue. Which is probably why its sellers are marketing it as such. However its capabilities in generating traffic is cut in half by the lower amount of TX queues. This is the reason to why the degradation factor k goes up when we reach higher throughput, the hardware is simply not designed for it, even though the design specifications told otherwise. The details of our evaluation is presented in Section 6.1.3

In summary we can see through the test scenarios that the system performs really well until we reach a threshold. That threshold is due to a design flaw in the chosen hardware. An interesting thought would then be to see if we could reach the designated packet rate by just doubling the amount of nodes. However since we are

already using all the nodes that was bought for the thesis this could not be tested. Even if it were to work it would make the system cumbersome to use. The test user expects to receive the throughput that is provided to the system, not half of it.

6.1.3 SBC hardware evaluation - finding the hard limit

To get some insight into what each single-board computer actually could perform, and for troubleshooting the solution, a small comparison was made on the hardware. The tests were made using `tcpreplay`, which is why the numbers are fluctuating as `tcpreplay` is a user-level application. This results in a lot of overhead when the packet size go down because there are so many interrupts and context switches. The reason to why `tcpreplay` was used is out of comfort reasons. Since it takes time to compile new kernels for each hardware just to enable `pktgen`.

The first comparison between the Banana Pro (BPro) and the Raspberry Pi (RPi) can be found in Table 6.6 and it can easily be concluded that the RPi is inferior when looking at the packet rate. It is not a surprise since the RPi's network interface only supports 100 Mbit when the BPro supports 1000 Mbit. As [20] tests in their paper, the most interesting comparison vectors are the results of using the lowest packet size at the highest requested packet rate. This results in the network interface transmitting packets as fast as it possibly can, since with the smallest permitted size it takes much less time to process the packet. In other words these results are the fastest packet rate the board can produce packets when using `tcpreplay`.

Platform	Packet size	Number of packets	Megabit per second (Mbps)	Packets per second (pps)
RPi	1500	100 000	93	8100
	1500	10 000	85	7407
	1500	1000	46	4000
	64	100 000	4	8643
	64	10 000	4	7813
	64	1000	2	4167
BPro	1500	100 000	865	74 873
	1500	10 000	608	59 643
	1500	1000	51	3846
	64	100 000	20 to 50	42 000 to 100 000
	64	10 000	18 to 37	37 000 to 77 000
	64	1000	16 to 27	33 000 to 50 000

Table 6.6: Performance test, comparison between Raspberry Pi and Banana Pro.

As was discovered in Section 6.1.2 we saw a severe degradation when using the ODROID as hardware for the nodes. Therefore we also did a quick evaluation and compared it to the other 1000 Mbit single-board computer we had, the BPro. In Table 6.7 we can see the same behaviour as was observed during the evaluation of the system. The ODROID is actually slower when using large sized packets even though according to the specifications it should support 1000 Mbit. According to the user support forums this is a design decision in the hardware where the TX queues for the network interface are only 2 in comparison to the RX queues where it has 4. The NIC and CPU can handle the high packet rate. However it does not have the physical space for the bigger packets. This was not in the original technical

specification. So the Odroid can actually receive data up to 1000 Mbit but it cannot generate data up to 1000 Mbit because of the hardware limitation.

We then raise the question if we should change the hardware for the solution to the BPro. When inspecting the figures for the packet rates we can establish that the ODROID can keep a similar packet rate to the BPro when the packet sizes are small. The BPro can be concluded to be the absolutely fastest card, however since some of the kernel drivers seems to be sloppily written, the network interface sometimes go up and down for no apparent reason, the Odroid was chosen to use for the core of the system. It was the most stable platform and the fastest when stability was taken into account.

Platform	Packet size	Number of packets	Megabit per second (Mbps)	Packets per second (pps)
Odroid	1500	100 000	360	33 100
	1500	10 000	224	19 607
	1500	1000	444	3846
	64	100 000	47	96 000
	64	10 000	15	32 258
	64	1000	2	4500
BPro	1500	100 000	867	75 757
	1500	10 000	817	71 428
	1500	1000	476	41 500
	64	100 000	47	95 238
	64	10 000	40 to 44	83 000 to 91 000
	64	1000	12 to 16	25 000 to 33 000

Table 6.7: Performance test, comparison between Odroid and Banana Pro.

6.2 Traffic configuration

One of the design requirements in Section 4.1 was being able to configure all the relevant layers of the TCP/IP stack. For the developed solution *pktgen* was finally implemented as the underlying tool for generating traffic. However since it is a software written mainly with performance in mind, its configuration options are limited. With *pktgen* we can configure the fields that we need to set up basic tests. However setting up more complex traffic scenarios proved to be a bit problematic with the limited configuration options. To evaluate *pktgen* and see if we could use any alternative option we instigated an evaluation on similar software. In this section we go into detail how that evaluation was made and present graphs and tables to further illustrate why the software for transmission of Ethernet traffic was chosen.

6.2.1 Criterias

To be able to compare the software a set of criterias needs to be established. We extended the initial criterias of the design requirements for the purpose of finding a candidate. However during testing some other qualities were discovered that had not been thought of at the time of when the design requirements were set. The following list contains the criterias that were used for the comparison.

- Packets should be configurable at relevant layers of the TCP/IP stack.
- It should be possible to create packet streams that contain one or several packets.
- Streams need to be able to reach speeds of 1 Gbps.
- Given a packet rate the software should actually send in this rate.
- The software should be able to send packets from the minimum to maximum permitted size, 64-1500 bytes per packet.
- Being able to set up complex traffic scenarios to simulate real traffic.

6.2.2 Scapy/tcpreplay

Scapy was the first software that was considered for the system. It is a packet crafting software written in Python [21]. It is possible to create raw packets and configure every option for each relevant layer and the packet size is configurable. It is limited in its way to create streams, although it is possible to define how many packets that should be sent, we cannot configure sending packets in bursts. For sending packets at higher rates it starts *tcpreplay* as a subprocess to send the packets. Scapy is not designed to support large amounts of data or sending data fast. That is why *tcpreplay* is used under-the-hood. However tcpreplay is also a software that runs in user-space and thus decreases performance. It generates too much overhead with context switches and interrupts so it becomes taxing for the CPU to handle sending packets at high rates. It is faster than Scapy, but not faster than pktgen.

In addition it does not seem to implement a good algorithm for sending packets. The actual packet rate often differs quite much in comparison to what the user would put in. We discovered this by doing a small test and comparing it to *pktgen*. In Figure 6.5 we present the results that shows this behaviour. This behaviour along with overall slow packets per second rate makes Scapy/tcpreplay unattractive. However what Scapy does really well is its simplicity in configuring the contents of the packets. It offers all kinds of customisation, even up to the application layer. If more specific customisation for the contents of a packet is more important than performance, then Scapy is a good choice.

6.2.3 Pktgen

Pktgen is a kernel module for Linux that is developed for performance. As [19] shows it is a powerful packet generator that works in the kernel space and is directed via the proc file system [19]. It is via the proc system we configure packets and traffic streams. If we look at the design requirements in Section 4.1 we are able to configure all the relevant fields of the Data Link, Network and Transport layer. Mainly ip addresses, MAC addresses and VLAN tags, however not much more. The streams can be configured to send in burst, to use delays, send both fixed number of packets and sending indefinitely until interruption by the SIGINT signal. The most interesting feature of pktgen is that it runs in kernel space, which means that the overhead is significantly reduced, resulting in much higher packet rates and also much more accuracy between the packet rate given by the user and the packet rate that the system actually outputs. We present a small comparison between *Scapy*

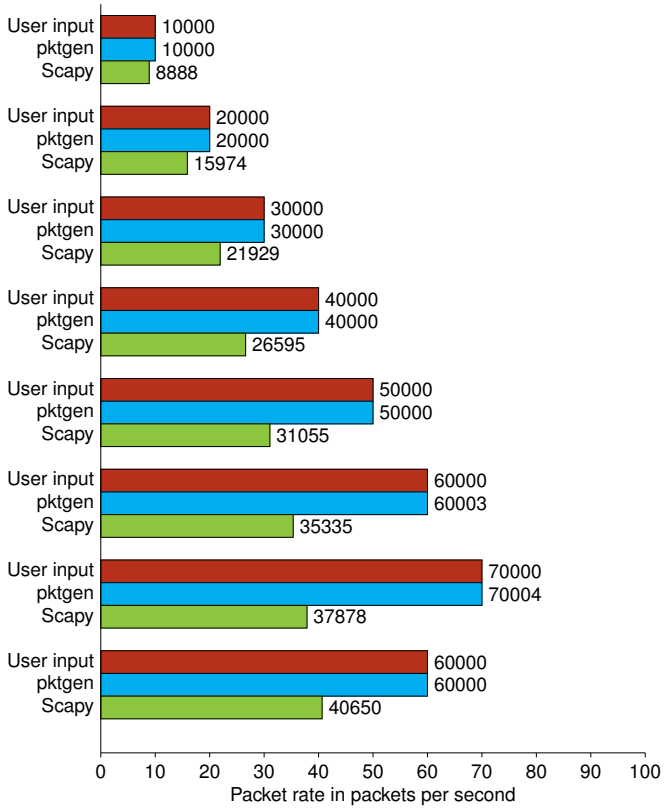

Figure 6.5: Actual outputs of the software with a given packet rate (pps).

and *pktgen* in Figure 6.5

In summary *pktgen* provides us with the tools we need both when it comes to packet configuration but especially with the stream of packet configuration. Another significant benefit is the pure performance of *pktgen*. It is not as easy-to-use as *Scapy* and it does not cover as many configuration options for the contents of the packets. However with the superior performance and the packet stream configuration it still wins over *Scapy*. However if we would have gotten requirements of more detailed packet configuration then the choice would not have been as simple.

6.2.4 tcpreplay

We considered using *tcpreplay* by itself since Scapy was using it under-the-hood. However it is, as the name suggests, reliant on replaying pcap files. So we have no ability in customising and configuring the contents of the traffic in an easy way. The application is also run in user-space so we are missing out on the performance brought by *pktgen*. It can generate packets at speeds higher than *Scapy*. However it does not come close to *pktgen*. If we ever were to capture traffic and replay it again, then *tcpreplay* would do a good job.

6.3 Packet capturing

In this section we look at the packet capturing capabilities of the system. We set up a test scenario and look at how high we can pressure the system before we see packet loss. One of the hard requirements was that we could send traffic without any packet loss at all, which is why we want to find this hard limit in the system. We begin by presenting the test methodology, we look at the criterias on the system and look at the results from the test. Finally we summarise and end the section with thoughts on the solution.

6.3.1 Test methodology

For the packet capturing test we made the test method even simpler. The goal is the same as for the evaluation of the transmitting side. We want to pressure the system and see if we can find the bottleneck, or the hard limit of how fast we can send packets and still capture them without any packet loss. We set up the test with just one receiving node and two transmitting nodes. Because as we discovered in Section 6.1.2 the ODROID only has 2 TX queues and thus only a 50% capacity for generating traffic. However it still has 4 RX queues so it can receive traffic much faster than it can generate. Therefore we used double the amount of nodes as transmitters, to try and pressure the ODROID to the max. The reason to why we need to find this hard limit, i.e. when the system starts dropping packets, is because otherwise we cannot know if it was the device under test that had packet loss or the system in itself. One extra node was setup as the master node to orchestrate the tests. Figure 6.6 shows the setup. The test method we used can be summarised in a list:

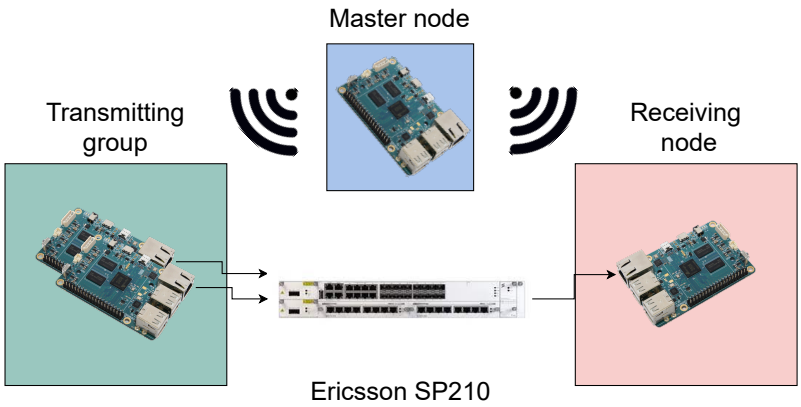

Figure 6.6: The capturing test setup.

1. One packet stream per node
2. Run a total of 1 000 000 packets
3. Compare captured packets with how many that were sent.
4. If no packet loss was discovered repeat the test for 10 times to ensure no packet loss.

6.3.2 The test scenarios - finding the zero packet drop rate

In order to try and find the hard limit we focused on what we believed were the two most taxing scenarios for the ODROID. Also taking inspiration from the testing in [20].

1. Largest packet size to reach highest throughput.
2. Smallest packet size to reach highest packet rate.

We started by testing for the highest throughput and thus using 1500 bytes as the packet size. To reach 1 Gbps, the maximum throughput, we need to send 1500 bytes of packets up to around 83000 packets per second. To get a confidence for our test results we repeated each test ten times. We summarise the test results in Table 6.8 As the results display we have on packet loss on two of the scenarios. It is a very

Test no.	Rate (pps)	Rate (Mbps)	Size(Bytes)	Packets sent	Packets dropped	Drop rate (%)
1	83000	1000	1500	1000000	0	0,00%
2	83000	1000	1500	1000000	0	0,00%
3	83000	1000	1500	1000000	0	0,00%
4	83000	1000	1500	1000000	0	0,00%
5	83000	1000	1500	1000000	214	0,02%
6	83000	1000	1500	1000000	0	0,00%
7	83000	1000	1500	1000000	0	0,00%
8	83000	1000	1500	1000000	614	0,06%
9	83000	1000	1500	1000000	0	0,00%
10	83000	1000	1500	1000000	0	0,00%

Table 6.8: Scenario 1: Large packet size - high throughput

small packet loss, but we still need to achieve the zero packet drop rate, it does not matter even if it is a very small amount of the packets that are dropped.

The next test scenario we performed was to use the lowest packet size instead and run the same test but with a higher packet rate. Hoping that we would achieve zero packet rate by using a smaller packet size. The test results are displayed in Table 6.9

Test no.	Rate (pps)	Rate (Mbps)	Size(Bytes)	Packets sent	Packets dropped	Drop rate (%)
1	90000	1000	64	1000000	0	0,00%
2	90000	1000	64	1000000	376	0,04%
3	90000	1000	64	1000000	0	0,00%
4	90000	1000	64	1000000	0	0,00%
5	90000	1000	64	1000000	114	0,01%
6	90000	1000	64	1000000	0	0,00%
7	90000	1000	64	1000000	445	0,05%
8	90000	1000	64	1000000	202	0,02%
9	90000	1000	64	1000000	362	0,04%
10	90000	1000	64	1000000	0	0,00%

Table 6.9: Scenario 2: Low packet size - high packet rate

To our big surprise we saw an even higher packet loss rate when using the smaller size and a little bit higher packet rate. This seems to suggest that the packet loss is not related to the packet size, but rather the packet rate. Thus we continued lowering the packet rate in the following tests. We reduced the packet

rate by 10000 packets per second for every scenario. The test results can be found in Appendix A. However instead of presenting them in a long table we used the results to produce a graph. Figure 6.7 presents our findings. The figure depicts a graph that measures how likely the system is to drop packets given a packet rate. Since a zero packet drop rate is required it does not matter how many packets were dropped, more the fact that it actually happened. So the numbers are simply based on how many tests of a given packet rate resulted in packet loss. Each packet rate was tested 10 times, except for the latter packet rates that were tested up to 30 times just to ensure there were no packet drops. The graph does not take into account how many packets that were dropped. As can be noted in the graph, the system seems to be able to capture all packets without drops when transmitting in 50 000 packets per second. This is a hard limitation of how fast packets can be sent when using the same kind of hardware in both ends.

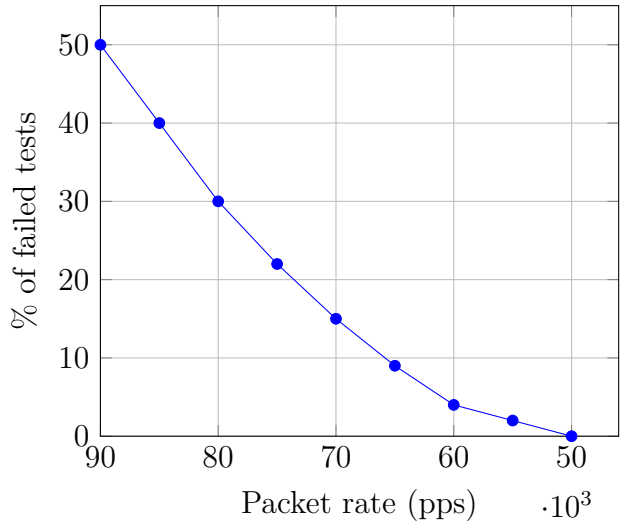

Figure 6.7: Packet rates and the percentage of failed tests.

6.4 Bottlenecks

In summary the developed solution is unwieldy to use. Designing precise, high-performant testing tools with general-purpose single-board computers is difficult. The technical specifications of the hardware does not live up to the actual performance of the board. Something that was very disappointing to find, in addition to finding the ODROID only had 2 TX queues, was the the 50 000 packet rate limit. This becomes a bottleneck for the performance of both the transmitting and receiving capabilities. This severely limits our ability to test high speed traffic. The system however can still be used alongside the proprietary tester, doing simpler tests. Replacing it fully though, that is not likely to happen.

Another bottleneck that was not really explored is the one of the Ericsson SP210 switch. During the work we never got the chance to use enough nodes to utilise all of its ports. With enough nodes we reach another problem of not having enough ports on the switch which forces us to either invest in more switch hardware or limit

the performance of the system to that of the number of ports we have in the switch. However the biggest bottleneck of the system still lies with the hardware flaw that we found in the ODROID C1 single-board computer.

7
Conclusion

In this thesis work we have designed and implemented a distributed traffic analyser as an alternative solution for the company Layer 10's traffic testers. The solution consists of a transmitting and a receiving group of distributed nodes and a master node that works like a handler for the system. The system is built using the MPI framework for communication between nodes. Pktgen for transmitting traffic and netsniff-ng for capturing traffic. Using these pieces in addition to the single-board computer ODROID-C1 we could build a system that worked as a traffic tester.

By testing this system we proved that it can generate and capture traffic at rates that fulfil the requirements of the company. However it is an unwieldy system to use due to its lack of real user interaction. Another big issue was the lack of full customization of the network traffic. Due to the limitations of pktgen, not that many different traffic scenarios could be set up. If we were to use Scapy instead, another software that comes with more customization options, we then get a serious performance hit. This lead us to offering two different solutions to the testing engineers. One that was performant but limited in its customization of traffic, or one where you could customize almost any field but then not as reliable performance-wise. However both these options were of interest to the test engineers as they could carry out limited testing and not be solely reliant on the proprietary traffic tester they already had.

Something to be noted is that we mostly evaluated the solution using traffic characteristics that were extreme. Such as very low packet sizes at a really high packet rate, or trying to achieve maximum wire speed. This in order to find the hard limits, or bottlenecks, of the system. Instead one could evaluate the system based on emulating real traffic characteristics. It was not considered since we realised that if we find the hard limits we know that we can send any sort of traffic as long as we stay within those limits. However if one were to evaluate the system based on real traffic scenarios it would be interesting to emulate that. To list a few emulations one could send traffic with an average packet size over a long period, or to emulate a population's use of Internet one could send more traffic during certain hours, the hours people are awake, over the course of a day.

Given the hardware evolution and possibilities for single-board computers today, a possible alternative to the ODROID-C1 could be any single-board computer that supports DMA (Direct Memory Access). This really enhances the performance of each individual card. We could also look at trying to evaluate the scalability of the system more. At the time we were only able to order so many nodes so we could not explore the scalability in a satisfying manner. We believe it would have been much more interesting to look at the scalability when we would have

reached an amount of nodes up towards twenty, or even higher. In addition a possibility would be of deploying the solution into the cloud. It would make the hardware configuration towards the physical radio unit a bit more complicated, but from a research standpoint much more interesting. Imagine just ordering another computing instance whenever you needed more performance in the packet generating or capturing.

Further improvements for future research could be made to the user experience of the system, making it easier to use. In addition to this, making the packet generating and capturing task agnostic. In other words, develop a framework for interpreting traffic configuration. Then translate that into configuration parameters for the underlying software. This could enable us to still use the same solution but under the hood use different software depending on what the user wanted to do. You could then extend the system with other packet generator and capturing software by writing integration APIs.

Bibliography

[1] K. Toshniwal and J. Conrad, "A web-based sensor monitoring system on a linux-based single board computer platform," in *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*, March 2010.

[2] J. Kiepert, "Creating a raspberry pi-based beowulf cluster," May 2013, accessed 2015-09-30. [Online]. Available: http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster_v2.pdf

[3] P. Ramanathan, K. Shin, and R. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, Oct 1990.

[4] K. Minghao, K. Y. Chyang, and E. K. Karuppiah, "Performance analysis and optimization of user space versus kernel space network application," in *2007 5th Student Conference on Research and Development*, Dec 2007, pp. 1–6.

[5] "Mpi: A message passing interface," in *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993, pp. 878–883.

[6] M. Group. (2017) Mpich overiew. Accessed 2017-01-30. [Online]. Available: http://www.mpich.org/about/overview/

[7] L. Dalcin. (2015) What is python? [Online]. Available: https://mpi4py.readthedocs.io/en/stable/intro.html#what-is-python

[8] (2010) Documentation, tcpdump & libpcap. Accessed 2015-09-30. [Online]. Available: http://www.tcpdump.org/

[9] (2010) Tcpreplay online manual. Accessed 2015-09-30. [Online]. Available: http://tcpreplay.synfin.net/wiki/manual

[10] (2010) What is iperf?, iperf.fr. Accessed 2015-09-30. [Online]. Available: https://iperf.fr/

[11] S. P. (2021) Features of ostinato. Accessed 2015-09-30. [Online]. Available: https://ostinato.org/features

[12] A. Botta, A. Dainotti, and A. Pescapè, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, 2012.

[13] C. Morariu and B. Stiller, "Dicap: Distributed packet capturing architecture for high-speed network links," in *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, Oct 2008.

[14] Anritsu. (2013) Network performance tester mp1590b. Accessed 2015-09-30. [Online]. Available: http://www.anritsu.com/en-GB/Products-Solutions/Products/MP1590B.aspx

[15] Spirent. (2015) About us. Accessed 2015-09-30. [Online]. Available: http://www.spirent.com/About-Us

[16] E. Wramner. (2014) Performance testing with a raspberry pi wall running java. Accessed 2021-11-12. [Online]. Available: https://www.jfokus.se/jfokus14/preso/Performance-Testing-with-a-Raspberry-Pi-Wall-running-Java.pdf

[17] G. Pemmasani. (2014) dispy: Python framework for distributed and parallel computing. Accessed 2015-09-30. [Online]. Available: http://dispy.sourceforge.net/index.html

[18] D. Borkmann. What is netsniff-ng? Accessed 2021-11-12. [Online]. Available: http://netsniff-ng.org/faq.html#g0

[19] R. Olsson. (2005) pktgen the linux packet generator. Accessed 2021-11-12. [Online]. Available: http://www.cs.columbia.edu/~nahum/w6998/papers/ols2005v2-pktgen.pdf

[20] L. Hassnawi, R. Ahmad, A. Yahya, S. Aljunid, and Z. Ali, "Packet size and packet rate effects over motorway surveillance system network performance," 04 2013.

[21] P. Biondi and the Scapy community. (2021) Scapy: Packet crafting for python2 and python3. Accessed 2021-11-12. [Online]. Available: https://scapy.net/

A

Appendix A

```
from mpi4py import MPI
import sys

if __name__ == '__main__':
    capture_args = ['capture_node.py']
    capture_info = MPI.Info.Create()
    capture_info.Set('hostfile', 'hosts/caphosts')
    capture_comm = MPI.COMM_WORLD.Spawn(sys.executable, args=
    ↪ capture_args, info=capture_info, maxprocs=3)

    # First barrier for capturing, makes sure each node is ready to
    ↪ capture.
    capture_comm.Barrier()

    transmit_args = ['transmit_node.py']
    transmit_info = MPI.Info.Create()
    transmit_info.Set('hostfile', 'hosts/transmithosts')
    transmit_comm = MPI.COMM_WORLD.Spawn(sys.executable, args=
    ↪ transmit_args, info=transmit_info, maxprocs=3)

    # Make sure each transmitting node is done transmitting.
    transmit_comm.Barrier()
    # Release each capturing node from barrier synch since every
    ↪ transmitting node is done.
    capture_comm.Barrier()

```

Listing A.1: Master node algorithm

```python
from mpi4py import MPI
import time

from gencapfw import PacketCapturer

if __name__ == '__main__':
    master_node = MPI.Comm.Get_parent()
    receiv_comm = MPI.COMM_WORLD()

    pc = PacketCapturer("eth0")
    pc.add_option("-s")
    pc.add_option("-H")
    pc.add_option("--ring-size", "500MiB")
    pc.start()
    while True:
        if pc.capture.readline().startswith("Running!"):
            receiv_comm.Barrier() # Barrier within recv group to make sure all have started their packet capturers.
            master_node.Barrier()
            break

    master_node.Barrier() # Wait for the sending nodes to finish, Barriers with master node.
    print("PC: Stopping packet capturer.")
    time.sleep(4)
    pc.stop()
```

Listing A.2: Receiving group software implementation

```
from gencapfw import PacketGenerator
from mpi4py import MPI

if __name__ == '__main__':
    master_node = MPI.Comm.Get_parent()
    comm = MPI.COMM_WORLD()
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        configs = []
        for i in range(size):
            conf = ["pkt_size 64", "dst 192.168.0.1",
            "dst_mac 00:1e:06:c2:11:13", "count 100000",
            "clone_skb 100000", "ratep 50000"]
            configs.append(conf)
    else:
        configs = []

    recvmsg = comm.scatter(configs, root=0) # Scatter the config to
    ↪ each node.
    pg = PacketGenerator()
    pg.loopset(recvmsg)

    comm.Barrier() # Synchronise each node to start at the same time.
    pg.start()
    master_node.Barrier()  # Make sure each node is done transmitting.
    master_node.Disconnect()  # Disconnect from master_node.
```

Listing A.3: Transmitting group algorithm

B

Appendix B

Test no.	Rate (pps)	Rate (Mbps)	Size (Bytes)	Packets sent	Packets dropped	Drop rate (%)
1	83 000	1000	1500	1 000 000	0	0.00%
2	83 000	1000	1500	1 000 000	0	0.00%
3	83 000	1000	1500	1 000 000	0	0.00%
4	83 000	1000	1500	1 000 000	0	0.00%
5	83 000	1000	1500	1 000 000	214	0.02%
6	83 000	1000	1500	1 000 000	0	0.00%
7	83 000	1000	1500	1 000 000	0	0.00%
8	83 000	1000	1500	1 000 000	614	0.06%
9	83 000	1000	1500	1 000 000	0	0.00%
10	83 000	1000	1500	1 000 000	0	0.00%
11	90 000	43	64	1 000 000	0	0.00%
12	90 000	43	64	1 000 000	376	0.04%
13	90 000	43	64	1 000 000	0	0.00%
14	90 000	43	64	1 000 000	0	0.00%
15	90 000	43	64	1 000 000	114	0.01%
16	90 000	43	64	1 000 000	0	0.00%
17	90 000	43	64	1 000 000	445	0.05%
18	90 000	43	64	1 000 000	202	0.02%
19	90 000	43	64	1 000 000	362	0.04%
20	90 000	43	64	1 000 000	0	0.00%
21	80 000	38	64	1 000 000	0	0.00%
22	80 000	38	64	1 000 000	1546	0.16%
23	80 000	38	64	1 000 000	0	0.00%
24	80 000	38	64	1 000 000	0	0.00%
25	80 000	38	64	1 000 000	0	0.00%
26	80 000	38	64	1 000 000	0	0.00%
27	80 000	38	64	1 000 000	0	0.00%
28	80 000	38	64	1 000 000	848	0.09%
29	80 000	38	64	1 000 000	0	0.00%
30	80 000	38	64	1 000 000	6694	0.67%
31	70 000	33	64	1 000 000	312	0.03%
32	70 000	33	64	1 000 000	0	0.00%
33	70 000	33	64	1 000 000	0	0.00%
34	70 003	33	64	1 000 000	0	0.00%
35	70 003	33	64	1 000 000	0	0.00%
36	70 003	33	64	1 000 000	0	0.00%
37	70 003	33	64	1 000 000	0	0.00%
38	70 001	33	64	1 000 000	0	0.00%
39	70 000	33	64	1 000 000	0	0.00%
40	70 000	33	64	1 000 000	0	0.00%
41	70 000	33	64	1 000 000	0	0.00%
42	70 000	33	64	1 000 000	0	0.00%

43	70 000	33	64	1 000 000	0	0.00%
44	70 000	33	64	1 000 000	0	0.00%
45	70 003	33	64	1 000 000	0	0.00%
46	70 000	33	64	1 000 000	268	0.03%
47	70 000	33	64	1 000 000	0	0.00%
48	70 000	33	64	1 000 000	0	0.00%
49	70 000	33	64	1 000 000	0	0.00%
50	70 000	33	64	1 000 000	188	0.02%
51	60 002	28	64	1 000 000	0	0.00%
52	60 002	28	64	1 000 000	0	0.00%
53	60 002	28	64	1 000 000	0	0.00%
54	60 002	28	64	1 000 000	0	0.00%
55	60 002	28	64	1 000 000	0	0.00%
56	60 002	28	64	1 000 000	0	0.00%
57	60 002	28	64	1 000 000	0	0.00%
58	60 002	28	64	1 000 000	0	0.00%
59	60 002	28	64	1 000 000	0	0.00%
60	60 002	28	64	1 000 000	0	0.00%
61	60 002	28	64	1 000 000	0	0.00%
62	60 002	28	64	1 000 000	0	0.00%
63	60 002	28	64	1 000 000	714	0.07%
64	60 002	28	64	1 000 000	0	0.00%
65	60 002	28	64	1 000 000	0	0.00%
66	60 002	28	64	1 000 000	0	0.00%
67	60 002	28	64	1 000 000	0	0.00%
68	60 002	28	64	1 000 000	0	0.00%
69	60 002	28	64	1 000 000	0	0.00%
70	60 002	28	64	1 000 000	0	0.00%
71	50 000	24	64	1 000 000	0	0.00%
72	50 000	24	64	1 000 000	0	0.00%
73	50 000	24	64	1 000 000	0	0.00%
74	50 000	24	64	1 000 000	0	0.00%
75	50 000	24	64	1 000 000	0	0.00%
76	50 000	24	64	1 000 000	0	0.00%
77	50 000	24	64	1 000 000	0	0.00%
78	50 000	24	64	1 000 000	0	0.00%
79	50 000	24	64	1 000 000	0	0.00%
80	50 000	24	64	1 000 000	0	0.00%
81	50 000	24	64	1 000 000	0	0.00%
82	50 000	24	64	1 000 000	0	0.00%
83	50 000	24	64	1 000 000	0	0.00%
84	50 000	24	64	1 000 000	0	0.00%
85	50 000	24	64	1 000 000	0	0.00%
86	50 000	24	64	1 000 000	0	0.00%
87	50 000	24	64	1 000 000	0	0.00%
88	50 000	24	64	1 000 000	0	0.00%
89	50 000	24	64	1 000 000	0	0.00%
90	50 000	24	64	1 000 000	0	0.00%
91	50 000	24	64	1 000 000	0	0.00%
92	50 000	24	64	1 000 000	0	0.00%
93	50 000	24	64	1 000 000	0	0.00%
94	50 000	24	64	1 000 000	0	0.00%
95	50 000	24	64	1 000 000	0	0.00%
96	50 000	24	64	1 000 000	0	0.00%
97	50 000	24	64	1 000 000	0	0.00%
98	50 000	24	64	1 000 000	0	0.00%

99	50 000	24	64	1 000 000	0	0.00%
100	50 000	24	64	1 000 000	0	0.00%

Table B.1: Table showing all tests made to find the zero packet drop rate.