# Bachelor Graduation Project

## Communication Architecture for a Hydraulic Simulation System

Sam de Jong & Jorden Kerkhof

4168941 & 4232461

Bachelor Thesis

TU Delft Delft University of Technology

# Bachelor Graduation Project

## Communication Architecture for a Hydraulic Simulation System

Bachelor Thesis

Sam de Jong & Jorden Kerkhof
4168941 & 4232461

13th of July 2017

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology

Supervisors:
dr.ir. Nick van der Meijs
ir. Herman Rave



TU Delft Delft University of Technology

HedoN Electronic developments

# Executive Summary

HedoN is a high-tech electronic development and production company that tasked a group of six TU Delft Electrical Engineering bachelor students to design a simulation system for hydraulic plants. This simulation system consist out of a central Hydraulic Simulation Unit (HSU) that runs a hydraulics simulation and controls various electrical components. This thesis describes the Ethernet communication between that HSU and the electrical components. (see Chapter 1)

As per Requirement 4, the communication system has to be designed to operate on a 1 millisecond tick. Every millisecond the HSU communicates new data with the various electrical components. This data consists of currents, voltages and operation modi for the components. (see Chapter 2)

Various Ethernet protocols were considered and in the end raw Ethernet communication on the OSI data link layer was selected. This is the most lightweight possible communications protocol through Ethernet to ensure the communication is fast enough and lightweight on processing power. (see Chapter 3)

Every millisecond the HSU broadcasts a big packet with all the data to every component. These components will then extract information relevant to their operation. This design choice moves the computational effort of selecting relevant data per component from the HSU to the components. This is desirable since the HSU needs enough processing power to run the simulation as well as the communication in parallel. The components will then process the data and provide their response with unicast packets. (see Chapter 3)

For every component to know exactly what data from the broadcasted packet is relevant to them, an initialisation phase is required. This phase is executed before the 1ms tick communication and ensures the components know at what offset they can find their relevant data. (see Chapter3)

This communication system was then tested to check if it does meet the 1ms requirement (see Chapter 4). It turns out the communication has some issues achieving the 1ms Requirement 4. The problem could be the non-realtimeness of the Linux kernel running on the HSU which leads to packets not meeting the 1ms requirement. A possible solution to this problem would be to program the HSU on a realtime OS or even baremetal C. (see Chapter 5)

# Contents

# 1 Introduction

## 1.1. Background Information

HedoN is a high-tech electronic development and production company based in Delft. The company was founded in 1979 as a spin-off from Delft University of Technology. HedoN has been Huisman's technology partner since 1979 for hydraulic motor controls. Huisman constructs large hydraulic plants that are controlled by a Hydraulic Motor Controller (HMC), which is designed by HedoN. This HMC handles input and output signals to and from the plant such that it can be operated in a safe manner.

Hydraulic plants need to be commissioned by very specialised commissioning engineers. Huisman has a small team of those engineers, but is interested in expanding that team. The problem with training commissioning engineers however, is that training can only happen at sea, where the hydraulic plants are. A solution to this problem would be to simulate the behaviour of these hydraulic plants. With a simulation system, commissioning engineers could be trained on land which greatly reduces training costs and in turn enables Huisman to afford more specialised commissioning engineers. Furthermore, a simulation provides a safer environment for commissioning engineers in training to make mistakes.

Another application of the simulation system could be for the engineers designing the HMC at HedoN. Right now HMC05 has been developed, but software fine-tuning based on results from a simulation could still be very helpful to improve the HMC. For example, the latest HMC supports up to four motors, where four HMC's could be connected parallel and will be able to control up to 16 motors. There's not yet a real hydraulic plant with this many motors, therefore a simulation would be helpful to test such a system.

An important design consideration concerning the simulation system is that it has been designed with modularity in mind. This will enable HedoN to perfect its simulation system with Huisman and then potentially branch out to other industries. Automotive and aerospace industries for example could also benefit from such simulation systems.

## 1.2. Simulation System

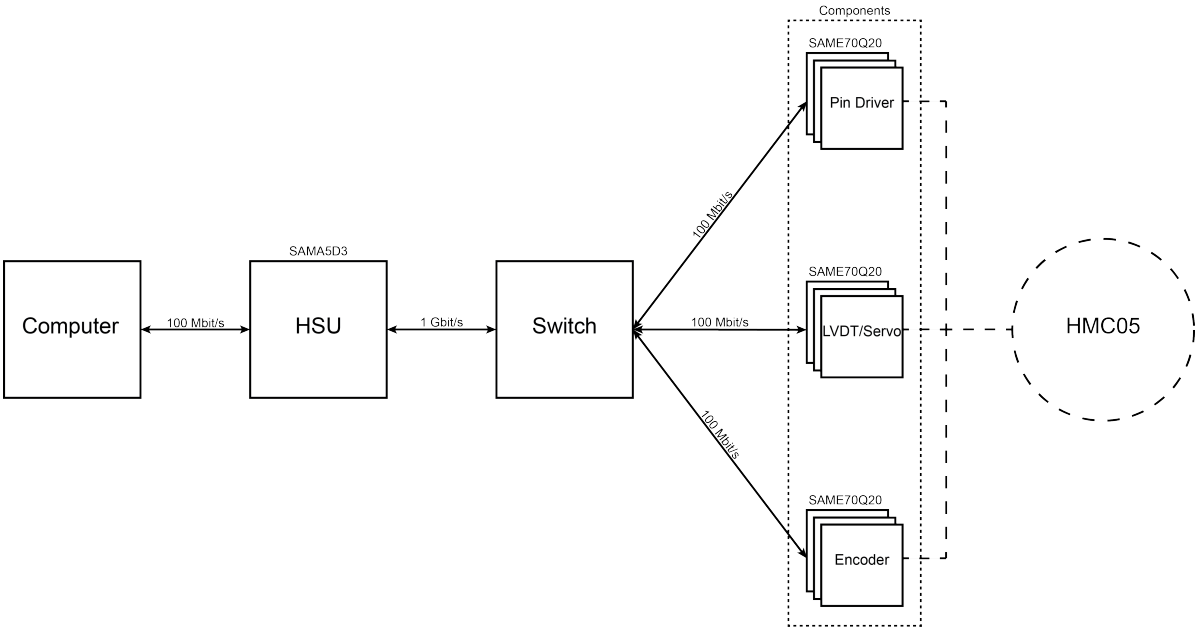


Figure 1.1: System Overview

In Figure 1.1 a schematic overview of the simulation system is shown. On the right hand side of the figure the HMC05, developed by HedoN, is displayed. This hydraulic motor controller would normally be connected to a hydraulic plant in order to control it. That plant has now been replaced by the simulation system.

In order to mimic the behaviour of a hydraulic plant, various electrical components are required to generate an analog signal for the HMC, namely pin drivers, LVDT's, servo's and encoders. These components then communicate through Ethernet with a Hydraulic Simulation Unit (HSU). The HSU runs hydraulic simulation code that resembles the dynamic behaviour of the hydraulic plant. Also, there is another Ethernet connection to an external computer/web-server to give relevant information to the operators of this simulation system.

As per assignment for the bachelor graduation project, the group of six students had to be split up in three sub groups. Group 1 will design the pin driver [1], group 3 will design the simulation on the HSU and a link to the computer/web-server [2]. Group 2 will design the Ethernet communication architecture, as will be described in this particular thesis.

## 1.3. Communication

This thesis describes the communication between the HSU and the components through Ethernet. The biggest challenge of the communication is to meet the requirement of a 1ms tick. Every millisecond the HSU needs to send new data to the components and receive a relevant answer. This time constraint exists because the HMC05 operates on a 1ms tick.

The scope of this thesis starts at the send and receive functions on the HSU and includes the switch. From there on it describes the software for the component running on a SAME70 micro controller up to the connection from the controller to the analog hardware. Furthermore it includes the full communication protocol/architecture and the initialization phase.

# 2

# Programme of Requirements

## 2.1. Contraints from the client

This bachelor graduation project has implementation constraints since certain requirements were simply already determined by HedoN. In order for this programme of requirements to make sense a series of implementation constraints first need to be listed.

Table 2.1: System Requirements

| Requirement number | Requirement explained |
|---|---|
| 1 | The simulation will be ran on a SAMA5 microprocessor, running a C++ code on a Linux kernel. |
| 2 | Every component will be controlled by a SAME70 microcontroller. |
| 3 | Ethernet will be used as communicaton method. |
| 4 | Communication cycle must be completed within 1 ms. |

**Requirement 1**
The simulation unit will be housing an Atmel SAMA5 microprocessor[3], since this microprocessor was readily available with a development kit. On the microprocessor a Linux kernel will run C++ code, this Linux kernel is provided by HedoN. A linux kernel was picked over baremetal C since programming in baremetal doesn't have libraries readily available on the SAMA5 and would thus take too long for a 10 week project. (See also[2])

**Requirement 2**
The components will be housing an Atmel SAME70 microcontroller[4] since this microcontroller was readily available at HedoN with a development kit. On the microcontroller baremetal C code will be running. Atmel provides a user friendly coding environment for programming the microcontroller called Atmel Studio.

**Requirement 3**
SPI and Ethernet were taken into consideration to use as communication protocol. Ethernet turned out to be more suitable for this system, because it is faster and Ethernet peripherals are available on the used microprocessor and microcontrollers.

**Requirement 4**
A full communication cycle must be done within 1 millisecond. A cycle starts at reading data from the memory and sending that data from the HSU to a component (Figure 1.1). The full cycle stops when data from the component has returned to the HSU and is written into the memory. So the roundtrip time of the communication should not exceed 1ms. This requirement is very important since the HMC is also operating on this same rate[5].

## 2.2. Design requirements

The HSU needs to send data to its components and receive data as a response, this data needs to be specified. Since the sub group responsible for hardware designed the pin driver, the specifications for the pin driver have been determined below. Implementation of these requirements in the communication can be found in Section 3.4

Communication from HSU to pin driver (see also[1]):

Table 2.2: Design Requirements - Data from HSU to Pindriver

| Requirement number | Requirement description |
|---|---|
| 5 | Output control signal |
| 6 | Current range control signal |
| 7 | External mode control signal |
| 8 | Value of output signal |

**Requirement 5**
The pindriver will have to set an output to a fixed value as voltage or current source. The output control signal will tell the pindriver which output is selected. Voltage has a range of 70 V (-35V to 35V), with a resolution of 1mV

**Requirement 6**
The current and the voltage on the pindriver are always measured. The current measurement can be done in three different ranges. The ranges are -35 mA to 35 mA, -350 mA to 350 mA and -3.5 A to 3.5 A with a resolution of respectively 1 $\mu$A, 10 $\mu$A and 100 $\mu$A.

**Requirement 7**
The pindriver is capable of connecting an external hardware component like a LVDT/servo or encoder for simulating a broken cable. So a selection has to be made to switch to this external modus.

**Requirement 8**
The value to which the output has to be set should be send to the pindriver.

Communication from pin driver to HSU (see also[1]):

Table 2.3: Design Requirements - Data from Pindriver to HSU

| Requirement number | Requirement description |
|---|---|
| 9 | Output control signal |
| 10 | Current range control signal |
| 11 | External mode control signal |
| 12 | Value of measured voltage |
| 13 | Value of measured current |
| 14 | Value of temperature with resolution of 1°C |

**Requirement 9**
The output control signal is sent back as acknowledgement for checking if the pindriver has the right configuration.

**Requirement 10**
The current range control signal is sent back as acknowledgement for checking if the pindriver has the right configuration.

**Requirement 11**
The external mode control signal is sent back as acknowledgement for checking if the pindriver has the right configuration.

**Requirement 12**
The pindriver is always measuring the voltage, that value has to be sent to the controller.

**Requirement 13**
The pindriver is always measuring the current, that value has to be sent to the controller.

**Requirement 14**
The pindriver is always measuring the temperature, which value has to be sent to the controller. The temperature on the board is measured with a resolution of 1°C and a range from -128°C to 127°C.

# 3

# Design

In this chapter a complete overview of the design of the communication system is described. The structure of the sections will represent the different parts needed to accomplish the full communication system. Firstly an overview of the general architecture is described in Section 3.1. In section 3.2 Ethernet will be discussed with details about the OSI-model. In section 3.3 the protocol will be explained. This is the protocol modified for this system where several design considerations are discussed. Section 3.4 will format the data which will be transferred between the HSU and the components. Datastructures are created and explained. The implementation of the protocol on the component controller and the HSU are discussed in 3.5.

## 3.1. General Architecture

In this communication system the HSU will be operating as the manager, because it runs the simulations and is keeping an overview of all components. The scope of this thesis is limited to the communication part for the HSU (for simulation part see [2]). A switch will be placed between the the HSU and the components to provide the physical connection. All the components consists of analog hardware and a microcontroller mounted on the PCB. The thesis also describes the implementation of the microcontroller and the integration with the hardware part (for hardware part see [1]). An overview of the architecture is given in Figure 3.1.

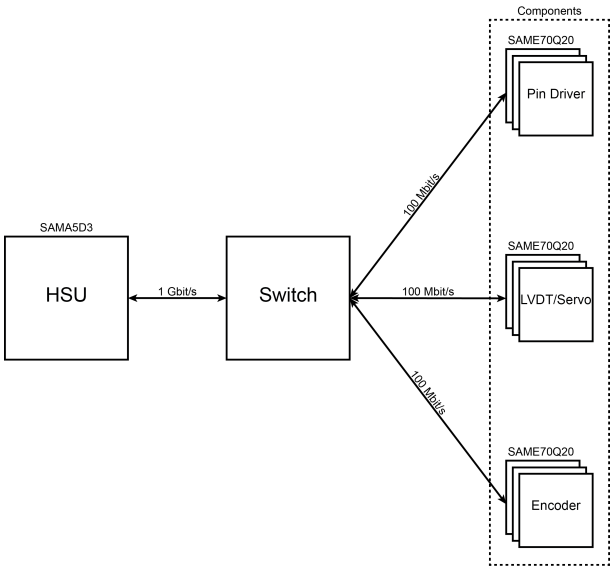


Figure 3.1: Architecture Overview

## 3.2. Ethernet and the OSI Model

As per Requirement 3, the components and the HSU will be communicating through Ethernet. In this section the protocol which will be used for communications will be selected. The OSI model(Table 3.1) is used as a

conceptual model to characterise and standardise communications. This model consists of seven abstraction layers with every layer serving the layer above it and is served by the layer below it[6]. The data being sent is thus accompanied by headers specified in every layer of the OSI model. Using this model, a communications protocol was decided upon.

Table 3.1: OSI Model

| **OSI Model** |
|---|
| Application layer |
| Presentation layer |
| Session layer |
| Transport layer |
| Network layer |
| Data link layer |
| Physical layer |

Since Ethernet is being used, the physical layer and data link layer of the OSI model are standard for Ethernet as defined by IEEE 802.3[7]. This is the international standard set for communications over Ethernet, consisting of a header with predefined parts. This header has a length of 18 bytes. On top of the datalink layer, higher layers of the OSI-model could be added. For these extra layers, various protocols are standardised which could be used.

Concerning Requirement 4, several protocols exists that can potentially be used to setup a reliable communication. TCP and UDP [8][6] for example are the two most well known communication protocols used for the internet on the transport layer. Both these protocols build on the IP protocol in the network layer. The Internet Protocol is used world-wide for huge networks to identify devices inside a (sub-)network. TCP and UDP are used for exchanging data on such a network.

A quick comparison between UDP and TCP is given below:

- User Datagram Protocol (UDP)
  - small header size (8 bytes)
  - connectionless with best effort delivery of packets
  - Packets may arrive out of order
- Transmission Control Protocol (TCP)
  - bigger header size (20 bytes)
  - three-way handshake and retransmission (TCP guarantees delivery of packets)
  - In order delivery

Moving further upwards on the OSI model all the way to the application layer, various other protocols arise. For industrial systems many protocols exist. Many of these protocols are based on TCP to be reliable with a low latency[9].

Application layer protocols have larger headers, and are mostly based on TCP so they also have a three-way handshake. Additionally, whenever a packet would not be received, retransmission would occur, which is undesirable on a system that consistently needs to communicate on a 1ms tick. Creating these headers takes processing power, which could otherwise be used for simulation. Furthermore the bigger the packet, the longer the transmission time. The reason to pick raw Ethernet over UDP is that the 8 bytes of UDP header and 20 bytes of IP header don't really do much in this case other than take up space. The communication is disconnected from the internet and will contain a relatively small network, so the communication will be limited to the data link layer of the OSI model. With raw Ethernet the most lightweight solution is found to help fulfil the 1ms tick requirement 4.

## 3.3. Communication Protocol

Only 2 layers of the OSI-model are used, because more layers result in more overhead which is unnecessary and will cause the communication to be slower due to the larger packet sizes. On this layer the Ethernet protocol is applying, which is the smallest possible protocol for Ethernet communication.

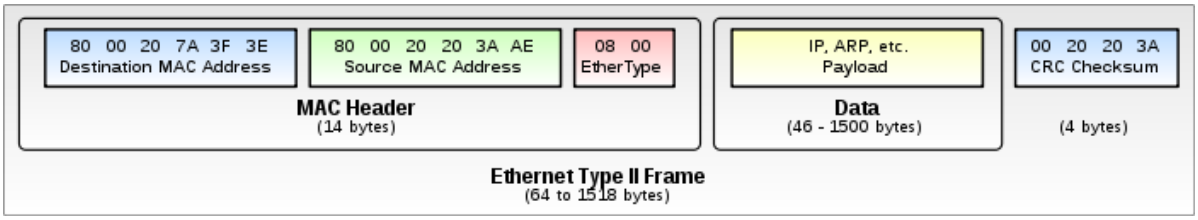


Figure 3.2: Ethernet Header (source: https://en.wikipedia.org/wiki/Ethernet_frame)

According to the IEEE 802.3 standard, raw Ethernet on the data link layer should have a header that is 18 bytes long[10]. This header (Figure 3.2) consists of:

- Destination MAC Address (6 bytes)
- Source MAC Address (6 bytes)
- Ethertype (2 bytes)
- Frame Check Sequence (4 bytes)

The destination and source MAC addresses define where the data is coming from and where it should be going based on the MAC address. MAC addresses are unique hardware addresses assigned to every device that has an Ethernet connection. The Ethertype states which protocol is being used in the higher layers of the OSI model. Since only raw Ethernet is being used, a special header for experimental communication was selected, 0x88B5[11].

Finally, the Frame Check Sequence (FCS) is a check sum that detects whether or not there were errors when transmitting a packet. If the check sum is correct, it is likely the data wasn't corrupted. If the check sum is incorrect, the data was corrupted during transmission and the packet is discarded. The discarding of the packet will be done by the switch[12]. So the HSU or the component won't recieve any packet. When the HSU or a component is not recieving any packet within 1ms, several errors could have occured. One of those errors is thus a failing FCS. At the moment errors can only be detected. As can be read in section 5.2.5, handling errors is discussed and still needs to be developed.

### 3.3.1. HSU to components communication

Since the protocol for communication has been selected, a deeper look needs to be taken into how the communication should proceed (see Figure 3.1) The simulation system runs from the centralised HSU on the SAMA5. This simulation should be able to send data to its various components that each house a SAME70. To relieve the HSU from some processing strain to send out many small packets, the HSU will be sending one big packet as a broadcast to all the components. Broadcast is achieved by setting the destination address in the Ethernet header to FF:FF:FF:FF:FF:FF. The HSU runs the simulation, so its processing resources should be allocated to that simulation as much as possible and not to communication.

The components will need to know what data in the big broadcast packet is relevant to them. To make sure they select the right data, two designs were taken into consideration.

**Indexed broadcast packet**

The first approach is to include an index in the broadcast packet indicating to every component where to look in the packet (Figure 3.3). For the sake of modularity this would be done by giving every component a custom MAC address, which is something that can be done since the simulation system operates offline. In this way the HSU does not need to learn the MAC-addresses and thus could the same HSU be implemented in any arbitrary simulation system. Every component could look for its MAC address in the index of the broadcasted packet and next to that MAC address an offset would indicate where relevant data can be found. The problem with this solution however is that the data sent as index is extra overhead in the packet. Furthermore, every component will take some time every 1ms tick to figure out what offset it has to use to read out relevant data.

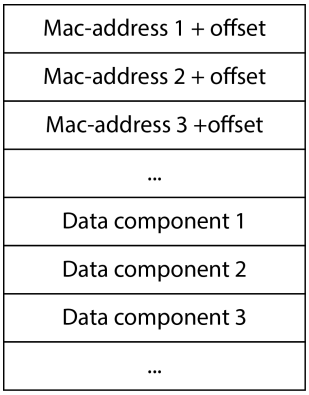


Figure 3.3: Indexed Broadcast

**Unindexed broadcast packet**

An alternative approach would be to just send all the data for all the components in a big Ethernet packet without any index (Figure 3.4) This can be done since there is another way to inform the components about what data is relevant to them. It could be accomplished with an initialisation phase (Section 3.3.3) that doesn't need to obey the 1ms tick. Within this initialisation phase the component will receive an offset from the HSU and will be using this offset for the further communication.


Figure 3.4: Unindexed Broadcast

**Conclusion**
The second approach decreases the length of the broadcast packet by leaving out the offset for every component in the system. Making the big broadcast packet smaller will increase the speed of the communication which is desirable to satisfy Requirement 4. Because of the initialisation phase, a steady connection is created between the HSU and components. Therefore the second option, a broadcast packet without an index, will be used for the communication protocol for this system.

### 3.3.2. Components to HSU communication
The various components also need to send data back to the HSU. This is done with a unicast since only the HSU needs this information and not the other components since they operate independently from each other. In order to send this information from the HSU to multiple components and sending this information back, a switch is required to connect them.

### 3.3.3. Initialisation
The initialisation phase ensures that every component is set up correctly before the actual simulation commences. When every component is initialised properly, the system will be ready to communicate on the 1ms tick. The important thing to note is that this initialisation phase is not dependent on this 1ms tick requirement.

Each packet sent will contain instruction codes. These instruction codes determine in which state the current system is. Each instruction sent has its predefined response, so the HSU and the components know what is recieved, what will be the next process and know what kind of packet with instruction code is expected next.

An overview of the instruction codes:

Table 3.2: Overview of instructions

| Instruction code | Explanation of instruction |
|---|---|
| 0x01 | The HSU requests a response from the component together with the component's required number of bytes. |
| 0x02 | The HSU sends out the offset for the component. The component will react by sending back this offset. |
| 0x03 | The HSU asks the component to send its offset. |
| 0x04 | This instruction will be used during the general communication. |
| 0x05 to 0xFF | Possible additional instructions. |

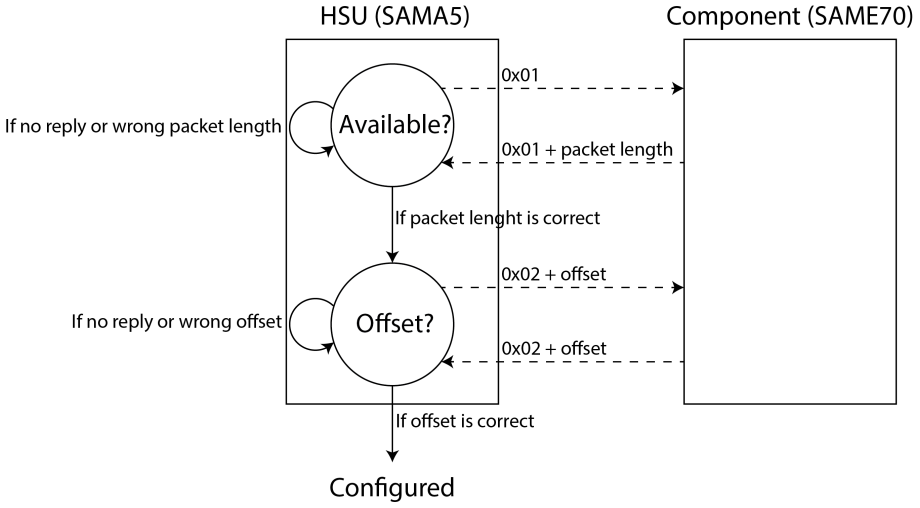


Figure 3.5: Initialisation Phase

An overview of the initialisation phase is shown in Figure 3.5. The dotted lines indicate communication between the HSU and the component and the solid lines indicate the hops between stages. The first packet sent from the HSU will have an instruction code at the beginning of the data sequence and is requesting an acknowledgement from the component. If the component receives that request it will send a reply back to the HSU together with a byte containing the number of the bytes the component wants to receive during the communication phase. The HSU already knows how many bytes the component wants to receive, but this answer is a good redundancy check to see if the communication proceeded correctly. If the HSU doesn't get a reply, or the component replies with the wrong number of bytes, the HSU will resend the first instruction code.

If the HSU correctly received the acknowledgement and the correct packet size, it will proceed to the next instruction. This instruction contains the offset the component will use to determine which part of the broadcast packet is relevant. The component will respond to this instruction by sending it back to the HSU. If the HSU recieves the offset from the component, the communication is ready to go. If wrong offset was received, or no response at all, the HSU will resend the offset. The HSU executes this procedure for every component simultaneously until every component is fully initialised so the communication on 1ms tick can commence [2].

After this initialisation phase, every component knows exactly where its relevant instructions in the broadcast packet are situated. Since the component will be defining the amount of data it wants by itself, it knows where the relevant data stops.

An addition is made to the broadcast packet, which is an instruction byte(0x04) at the beginning of the data packet. This is the instruction telling the component it is a communication packet and should thus be looking for relevant data in the broadcast packet with its given offset. Following this instruction, the component will capable of interpreting the data.

Another instruction byte(0x03) was implemented for future implementation. This instruction simply requests the offset from a component. In case a component is to misbehave, one of the first things the HSU could do is request its offset in order to judge what is wrong with the misbehaving component.

Also an extra specific instruction byte (0x05) could be used to for example specifically request the temperature of a component.

## 3.4. Data to communicate

### 3.4.1. Data from HSU to Pin Driver

To configure the pin driver, some settings need to be sent from the HSU. The pin driver operates in four different modi. It has to be able to deliver a fixed current or fixed voltage and be able to measure a current or voltage. The pin driver is only in one modus at the same time [1]. To explain the four different situations briefly:

- Sending a fixed current:
  The component controller (SAME70) has to send a signal where the current is set to a fixed non-zero value.
- Sending a fixed voltage:
  The controller has to send a signal where the voltage is set to a fixed non-zero value.
- Measuring current:
  The controller has to send a signal where the voltage is set to a value of '0 Volt'.
- Measuring voltage:
  The controller has to send a signal where the current is set to a value of '0 Ampere'.

So actually, two values have to be sent to the pin driver: which output has to be set (voltage or current) and which value this one has. Defining if the current or the voltage has to be set can be regulated by one bit. A '1' means setting the current and a '0' means setting the voltage. The value of the output will be sent as a IEEE754 single point float [13], to satisfy the desired resolution for the pindriver[1]. Also the pindriver always wants to know the range of the current, so accurate measurements can be done. There will be three different ranges (-35 mA to 35 mA, -350 mA to 350 mA, -3.5 A to 3.5 A) used by the pindriver, which will be represented by 2 bits in the data. The control signals will be sent to the pindriver by putting the signal on Input/Output pins implemented in the microcontroller.

The pindriver also has an external modus, which has to be controlled from the HSU, so another bit for this will be sent in the datapacket.

Table 3.3 gives an representation of the datastructure

Table 3.3: Overview of data from HSU to controller

| Data | Bitsize | Bytesize |
|---|---|---|
| Control signal for which output has to be set (current/voltage) | 1 bit | 1 byte |
| Control signal controlling in which current range the pindriver has to operate | 2 bits | |
| Control signal telling the pindriver if it is in external modus or not | 1 bit | |
| The value of the output which has to be set | N/A | 4 bytes |

Note that the control signals use a total of four bits. They will be sent as part of a byte, because a byte is the common format for sending data in communication protocols. The bandwidth of the Ethernet connection will be sufficient enough to leave 4 bits unused. Also this offers enough room to be able to expand the control signals, which increases the modularity of the simulation.

### 3.4.2. Data from pindriver to HSU

The Hydraulic Simulation Unit of course wants as much as information as possible to keep track of the components and can create an overview of the current status of the system. The pindriver always measures the voltage and the current of the output, also in sending mode. Setting the modus for the pindriver is needed for the hardware to deal with his input-/ output ports connected to the HMC. So these values of course will be sent back to HSU. These values will also be sent as a IEEE-754 single point float[13], just as the received value, to ensure the required resolution and not lose significance in the results.

The voltage and the current values are gathered from the pindriver through an ADC. As mentioned in section 3.4.1, the integrated ADC on the SAME70 doesn't have sufficient resolution for the pindriver. So an external

ADC is chosen with a SPI connection to the controller. As an extra check it is nice to send back the configured settings received from the HSU. This will be used at the HSU to check whether the pindriver is operating correctly, which means the received data is useful for the communication. These control signals will be sent back in the same method they're received, contained in one byte.

For safety of the pindriver, a temperature sensor is mounted on its PCB to measure the temperature of the power op-amp [1]. This is implemented by an NTC and thus can be read by an ADC and sent to the controller. Because the temperature needs a precision of 1 °C (Requirement 14), only one byte will be sufficient to send the temperature to the HSU. One byte can take 256 different values, which can meet the operating temperature range of the power op-amps.

Table 3.4 represents the datastructure for the data sent from the controller to the HSU.

Table 3.4: Overview of data from controller to HSU

| Data | Bytesize |
|---|---|
| Control signals for checking | 1 byte |
| The voltage measured at the pindriver | 4 bytes |
| The current measured at the pindriver | 4 bytes |
| The temperature measured at the pindriver | 1 byte |

## 3.5. Implementation

### 3.5.1. Implementation on the component controller

**Data structure**
After defining the protocol, the operating protocol needs to be implemented (coded) on the microcontroller. The code on the component controller is written in C. Atmel has created a special environment for programming its microcontrollers, Atmel Studio. This software environment has an Atmel Studio Framework (ASF)[14], where a lot of functions are pre-written for the ease of use. For simple implementation of the peripherals present on the SAME70 these functions are used. In this case especially the functions of the Ethernet port are used. For sending the packets and creating the ethernet headers, structures are defined. To show as an example, the structure Ethernet_header has the following members: destination address, source address and type. There are different structures for the headers and the layout of the data itself.

```
typedef struct ethernet_header {
        uint8_t et_dest[6];   /**< Destination node */
        uint8_t et_src[6];    /**< Source node */
        uint16_t et_protlen; /**< Protocol or length */
};
```

For implementation of the protocol itself two possible options were taken in consideration: a Finite State Machine and a Switch-case statement.

**Initial outline of a Finite State Machine**
The protocol uses an initialisation phase first before going to the communication of the system. This phase is step-based, so a FSM-like system, including two cooperating state machines, could be implemented by a state for each step in the initialisation. An overview of this initial outline of a possible FSM implementation is shown in Figure 3.6.

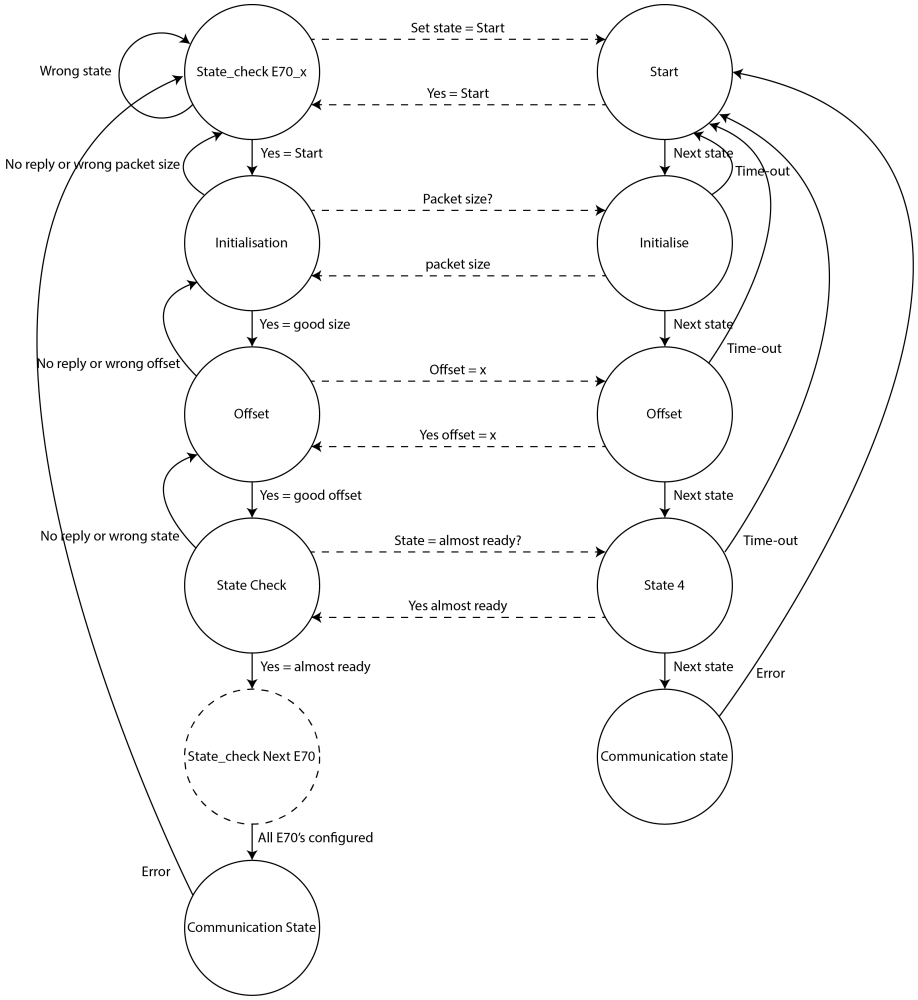


Figure 3.6: Initial outline of design of 2 state machines

The FSM concept starts at the "start" state of the HSU. The HSU will start with sending the first packet containing the first instruction code. This instruction code asks if the component is in the "start" state. The component will recieve this packet, send a reply and will go to next state. When the HSU recieves the answer, and it is correct it will go to the "initialisation" state. When he's receiving the wrong answer, the HSU will fall back to the previous state. In the second state the HSU will send the second instruction to the component and goes to the next state. When the component is receiving the correct instruction, it will send a reply containing the number of bytes it wants to recieve and goes to his second state. The HSU will receive the packet from the component and will check for the instruction code. If the instruction code is correct it will move to the "offset" state and send the next instruction together with an offset to the component. If the received packet is incorrect it will fall back to the previous state. When the component receives instruction three and the offset it will send back the offset as confirmation and move on to the next state. When the HSU gets a confirmation from the component with the correct offset it will move on to the last state: the communication state. Both the HSU and the component will stay in this state until an error occurs.

There will be 2 occasions an error could occur. The first occasion is when the device is not recieving any packet within a certain time. A time-out occurs and will force the device to go back to its "start" state. This fall-back hop to the "start" state after a time-out applies for the component at any particular state. The other error is caused by receiving wrong information, like a wrong instruction code. This will bring back the device back to the "start" state. Mention this is an exception for the HSU since in all other states it will just go to the previous state in the FSM.

**Switch-case statement**
The other way to implement the protocol is by programming a switch-case statement. Every packet sent by the HSU will at least contain the Ethernet header including the destination address, source address and type. When a packet is received a filter will take out the packets which are expected to arrive at the Ethernet port. First the destination address is checked, whether it is meant for the component or it is a broadcast packet. Other packets have no purpose for the component. When a personal packet is received, the instruction code will be checked with a switch-statement. Depending on which instruction code is sent, the component controller will process the data and respond with a proper reaction as mentioned in Section 3.3. The packets expected to be received and packets expected to be sent are independent on the previous packets processed. When a broadcast packet is received (so the destination address is FF:FF:FF:FF:FF:FF), the Ethernet type of the packet is checked. The type should be 0x88B5 as mentioned in Section 3.3. Also the instruction code should be 0x04. When these values are correct it is a communication packet and the data could be read and processed. Afterwards the component will send a packet with new data to the HSU. So every packet received is filtered and only when it is a correct packet, a process is started corresponding to the received data.

**Comparison**
Comparing the FSM to the switch-case statement, the FSM uses extra information which is not necessary. The instruction codes implemented in the packets are sent no matter if a FSM is used or not. By implementing a FSM, new variables are initiated to keep track of the states. The switch-case code for the component controller will operate fine. The could be seen as the master and will be the manager of the communication. So the HSU should keep track of states of all components, this is not necessary for the components themselves. Undesirable effects of the FSM occurs when an error occurs, both the systems have to fall back to the "start" states and set up a new connection. With the Switch-case system it is up to the HSU to restore the communication. The initialisation phase is the reliable part of the protocol, where the filters are for redundancy of the code.

Because the code is idle most of the time,the component controller will operate efficiently. The code is waiting for something to appear at the Ethernet port. When a packet is received, a function to process the packet is called. This process filters the packets and throws them away if they're not meant for the component. When a personal packet is received, an initialisation function is called. If a broadcast packet is received, the communication function is called. When there's no packet received on the Ethernet port, the controller will be idle. For easy understanding, the basic operation of the code is represented in a pseudo-code shown in Appendix A.1.

**Hardware implementation microcontroller to pindriver**
To communicate the values gathered from the HSU to the pindriver a DAC will be used. The SAME70 itself contains two analog-front-end controllers (AFECs) consisting of twelve channels and integrating an analog-to-digital converter (ADC), a programmable gain amplifier (PGA), a digital-to-analog converter (DAC) and two 6-to-1 multiplexers [4]. In this case the DAC would be very useful to use. Unfortunately the AFEC has a resolution of 12 bits, where 16 bits are needed. The SAME70 also includes one stand alone DAC with two channels. This DAC also has a resolution of 12 bits which will not be enough to satisfy the resolution needed. The resolution could be increased by dithering but this wouldn't be accurate enough and also requires more hardware. So the pindriver will have an external DAC (and also external ADC) integrated in the circuit. The SAME70 is supporting an $I^2C$ as well as a SPI connection, there are DAC's available with those connections. SPI is able to have a higher speed than $I^2C$, therefor a connection to the DAC will be done with SPI.

### 3.5.2. Implementation on the HSU [2]
From the start, the HSU will start processes and will set up a connection between the components and the HSU. The HSU knows from the start which components should be present so it can run the simulation it wants to run. It also knows how many bytes every component wants. By knowing this and the total amount of components, it determines the locations for the data in the broadcast packet. The HSU maintains a register where it keeps track of the statuses of the components. A simple visualisation of this register is shown in table 3.5 as an example.

Table 3.5: Example of the register of the components in the HSU at a particular moment in initialisation phase

| Component Number | Requested Bytes | Offset | Available | Offset Known |
|---|---|---|---|---|
| 1 | 10 | 0 | 0 | 0 |
| 2 | 10 | 10 | 0 | 0 |
| 3 | 12 | 20 | 1 | 0 |
| 4 | 10 | 32 | 1 | 1 |

In Table 3.5 a snapshot is presented of an example register at a particular moment in initialisation phase. The first process the HSU will start is the initialisation phase. It will do this for all components in parallel. At the beginning, all of the values for requested bytes and offset are already defined and known for the HSU[2]. The Available and Offset Known values start with a '0'. After sending the first instruction it will keep sending this until a correct reply is received. The correct reply contains instruction code '1' and the correct number of bytes requested. When this packet is received, the available bit will be set to a '1'. Next the HSU will keep sending the second instruction, asking the component to set its offset. After a proper reply from the component also the Offset Known bit will be set to '1'. So first it will try to check the available bit and then it will try to check the Offset Known bit. When both are a '1', the component is correctly initialised. The HSU will follow this procedure for every component. When every component is fully initialised, it will go to its next process. The next process is the communication phase. So after full initialisation it will stay in the communication phase and will run the hydraulic simulation.

To send and receive raw Ethernet packets, a freely available code from github was used [15]. This code was then altered such that it doesn't create new sockets every time a packet is sent but rather does that only once. Eventually this send and receive code was turned into a function and used as part of the simulation code running on the HSU written by group 3 [2].

# 4

# Evaluation

In this chapter the communication is tested to check if it meets the 1ms requirement. Section 4.1.1 describes the test setup used, in Section 4.1.2 this test is executed without task priority, in Section 4.1.3 with task priority and in Section 4.1.4 the influence of packet size is reviewed. Section 4.2 describes the communication when a simulation is running on the HSU and Section 4.3 describes the communication with a 1ms tick.

## 4.1. The 1ms Requirement

### 4.1.1. Test Setup

As per requirement, the simulation system needs to operate on a 1ms tick. In order to test this requirement, a test setup was devised:
In preparation: Initialise one component controller (SAME70).

- HSU (SAMA5) creates the broadcast packet
- HSU sends that packet to the component controller through Ethernet
- Component controller receives the packet and so does the PC running Wireshark
- Component controller creates a packet
- HSU receives the packet

This loop will be executed 10,000 times and analysed to determine if the communication system is fast enough. Whenever an answer from the component controller is received, a new packet is transmitted by the HSU. It will send its data through a switch as a broadcast packet. This means a PC running Wireshark can detect the broadcasted packets and calculate the time in between packets (Figure 4.1). This test does not test the 1ms tick, but rather the maximum speed that can be achieved through raw Ethernet communication. Whenever the PC receives a new packet, it knows the previous packet has travelled from the HSU to the component controller and back again, prompting a new packet to be sent.

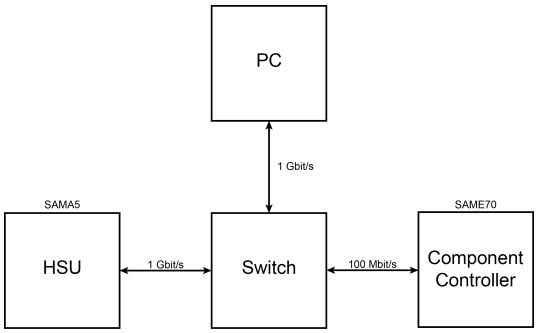


Figure 4.1: Test Setup

This particular test setup was selected because the simulation system wasn't capable of a few desirable features:

- Send data on a 1ms tick
- extract relevant information from the received packets
- Log incoming Ethernet packets with time stamps on the HSU

Therefore it was decided to run this test setup with Wireshark, using a packet size of 60 and sending data as fast as possible.

### 4.1.2. Unprioritised Communication

Initially, 10,000 packets with a size of 60 bytes were captured with Wireshark. But as can be seen in figure 4.2, some packets did not meet the 1ms requirement. Plotting this data on a logarithmic scale, illustrates that the majority of the packets did meet the requirements, but there definitely is an unacceptable amount of outliers.

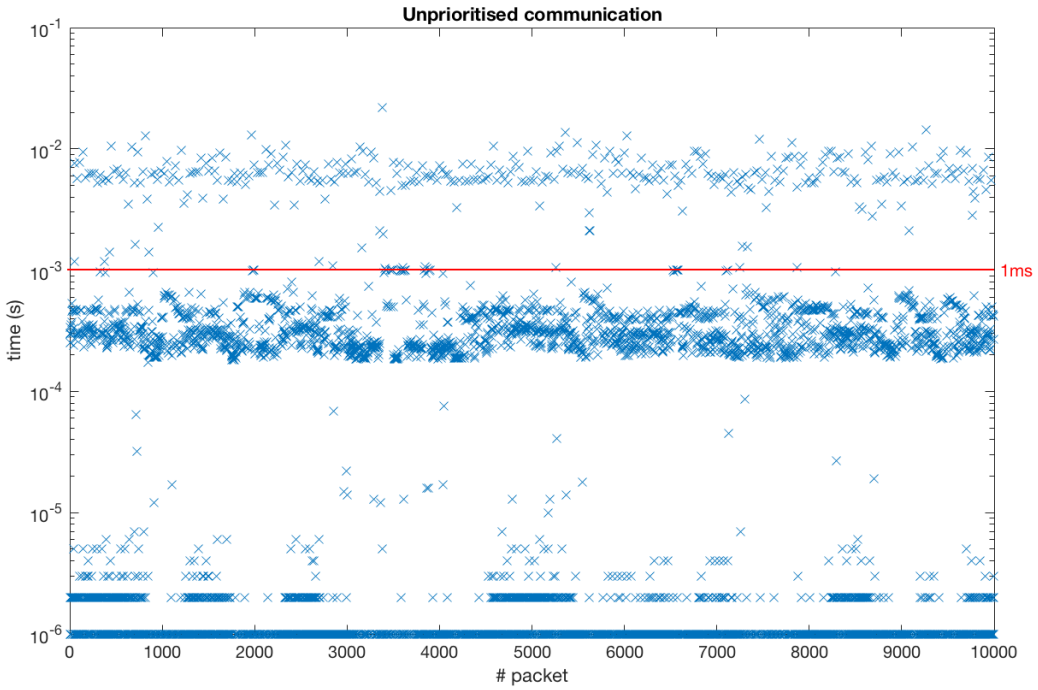


Figure 4.2: Unprioritised Communication

To get a better overview of the data, it was divided into five categories: (Figure 4.3, Table 4.1)

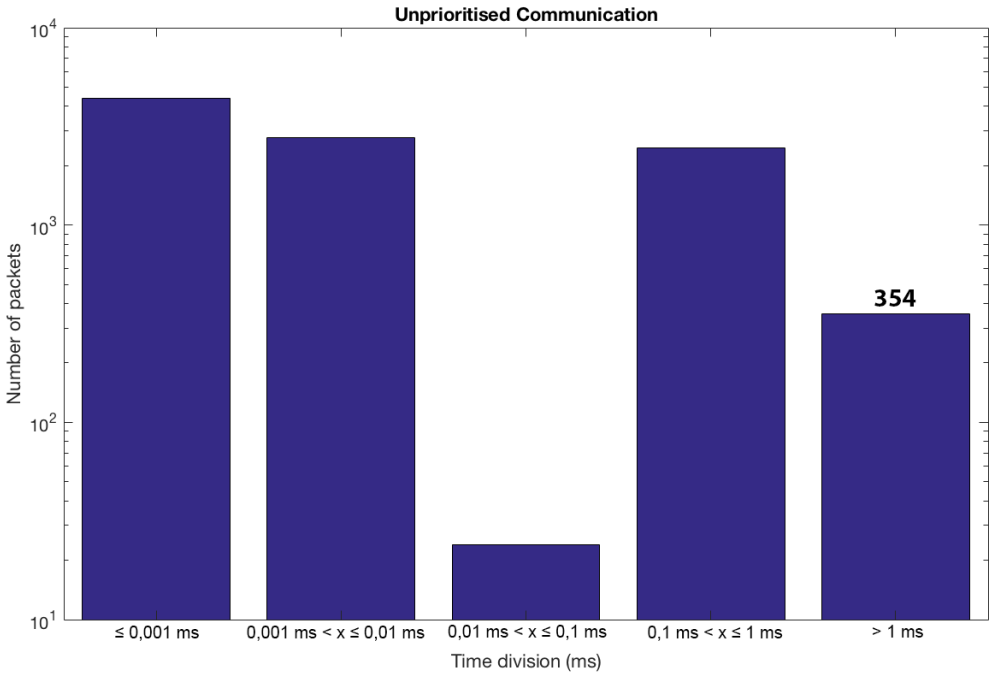


Figure 4.3: Unprioritised Communication Distribution

This shows that of the 10,000 packets being communicated, 354 do not meet the 1ms requirement (Table 4.1). It should be noted that the packets faster than 0.01ms can't have been measured accurately enough since at that point the switch buffering could influence the results. Industrial switches can have latency's in the order of magnitude of 0.01ms [16] and for consumer switches that can be even higher. For this test a TP-Link TL-SG105 [17] was used, for which no data about buffering or latency could be found. This means that at least the sub 0.01ms data might not be completely accurate, but that doesn't matter since those packets are fast enough anyways. The packets of interest are the ones that took over 1ms to be communicated, on which the switch latency is thought to have a negligible effect . Further information about the packets was gathered as can be seen in Table 4.2. On average the 1ms requirement is met and most packets go far below the 1ms requirement as can be derived from the median. There are unacceptable outliers however with a maximum of even 22 ms to communicate data.

Table 4.1: Distribution of Unprioritised Communication

| Time Range | Amount of Packets |
|---|---|
| x<=0.001ms | 4407 |
| 0.001ms<x<=0.01ms | 2761 |
| 0.01ms<x<=0.1ms | 24 |
| 0.1ms<x<=1ms | 2453 |
| x>1ms | 354 |

Table 4.2: Unprioritised Communication data

| | time |
|---|---|
| mean | 0.2993 ms |
| median | 0.0013 ms |
| max | 22.0510 ms |

Another interesting thing to note is that the median value is 0.0013ms. This is explainable because Wireshark has a resolution of 0.001ms and that means over half of the packets are actually faster than 0.001ms. The

switch might influence this result as well though, since theoretically a 0.001ms speed is impossible.

A speed of 0.001ms doesn't make any sense, because the theoretical maximum speed to send out 60 bytes (480 bits) over a 100 Mbit/s connection is 4.80 µs, which is larger than 0.001ms. So a possible explanation for this result is the latency of the switch. Luckily, the purpose of this test is not to measure the speed of the fast packets, but rather to analyse the packets that are too slow.

One explanation for this delay of the slow packets could be that a Linux kernel is not real-time. It's an operating system that has many other tasks to run, so it might not always be ready to communicate data in time. Therefore it was decided to alter the priority of the communications task on the HSU.

### 4.1.3. Prioritised Communication

The same test was executed again, but now by giving the process on the HSU a high priority. This was achieved by typing a command in the terminal that changes the priority of a task:

**nice -n -20 cranesim**

Linux uses niceness as a measure of priority where -20 is the highest priority and 19 the lowest priority[18]. The niceness has now been set to -20, which is the highest possible priority for a user to configure. As can be seen in Figure 4.4, this made a very significant difference since now a lot more packets do meet the 1ms requirement.

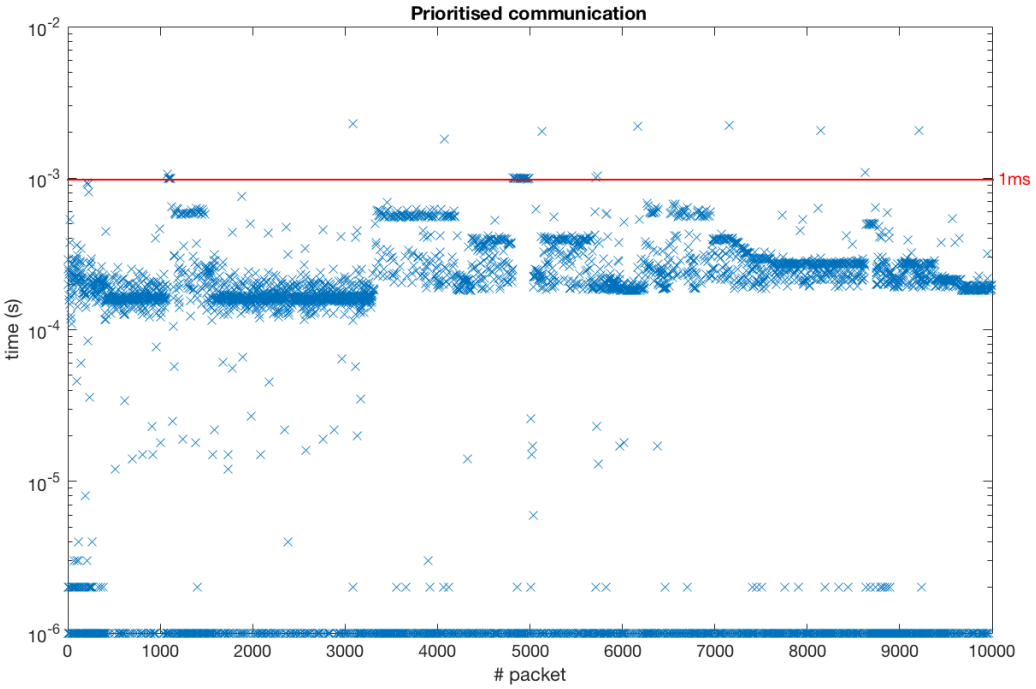


Figure 4.4: Prioritised Communication

To get a better overview of the data, it was divided into five categories again: (Figure 4.5, Table 4.3)

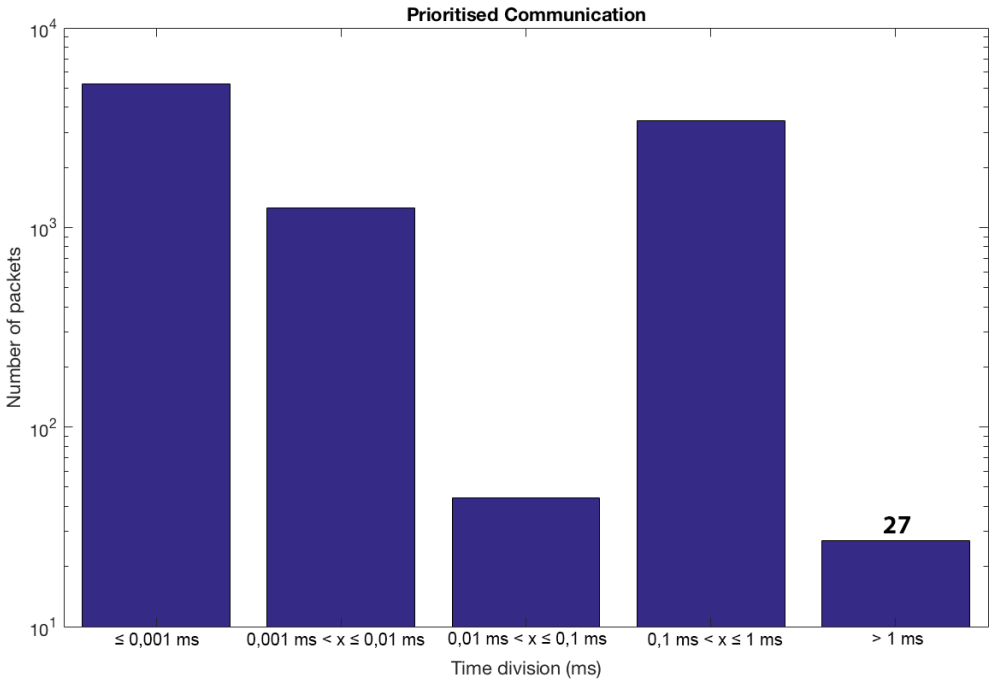


Figure 4.5: Unprioritised Communication Distribution

Out of 10,000 packets transmitted, 27 did not meet the 1ms requirement (Table 4.3).

Table 4.3: Distribution of Unprioritised Communication

| Time Range | Amount of Packets |
|---|---|
| x<=0.001ms | 5259 |
| 0.001ms<x<=0.01ms | 1249 |
| 0.01ms<x<=0.1ms | 44 |
| 0.1ms<x<=1ms | 3420 |
| x>1ms | 27 |

The prioritized process does however improve performance by a lot, which means it is very likely the packets that do take longer than 1ms do so because a Linux kernel is not real time. The average time taken by the packets is a little bit skewed since as was discussed before, the switch influences the accurate measurement of the packets with an order of magnitude of 0.01ms. But even when taking the influence of the switch into account, from an average communication speed of 0.0866, it can be concluded the communication is fast enough on average. This could indicate that if the Linux kernel was real-time, the communication would meet the 1ms requirement. From now on all future tests will be executed with the highest priority.

Table 4.4: Prioritised Communication data

| | time |
|---|---|
| mean | 0.0866 ms |
| median | 0.0010 ms |
| max | 2.2770 ms |

### 4.1.4. Packet Size Influence

In this section the influence of the size of the packets is tested. A small packet of 60 bytes is compared with a relatively big, arbitrarily chosen packet of 1014 bytes to determine if the packet size influences the performance.

It can be seen (Figure 4.6 and Figure 4.7) that the packet size doesn't change anything about the 1ms tick requirement. Communicating 10,000 packets of 60 bytes results in 8 packets that take too long and with 1014 bytes there are 10 packets that take too long. The average time the 10,000 packets take to be communicated however does increase with packet size. The packet of 60 bytes takes on average 0.088832 ms to be communicated compared to the average of 0.11259 ms for the packet of 1014 bytes. So while bigger packet sizes obviously do increase the time it takes to communicate, the possible non-realtimeness that leads to slow packets from the HSU is of much bigger concern. Judging from the averages, the communication would still meet the 1ms tick requirement with the bigger packet size.

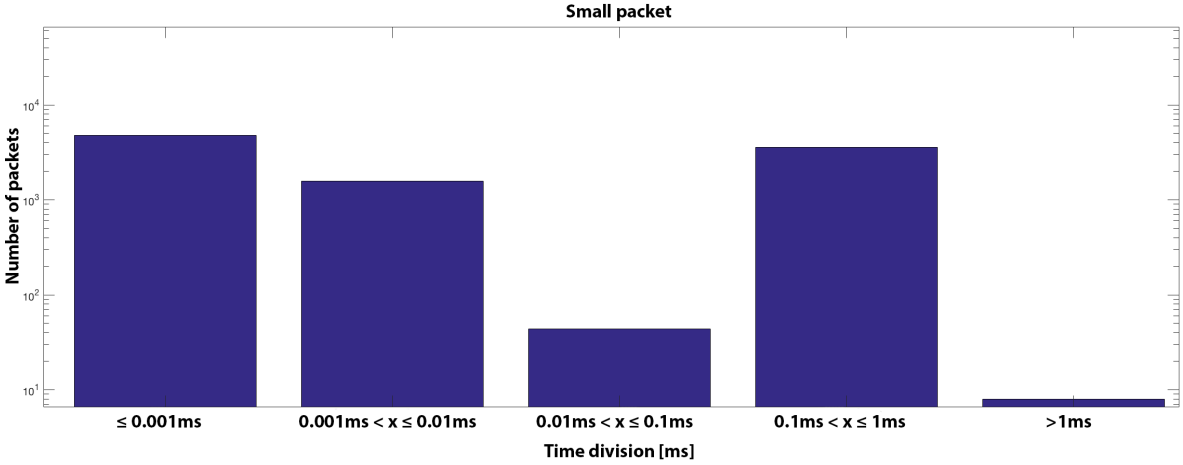


Figure 4.6: Communication of a small packet (60 bytes)

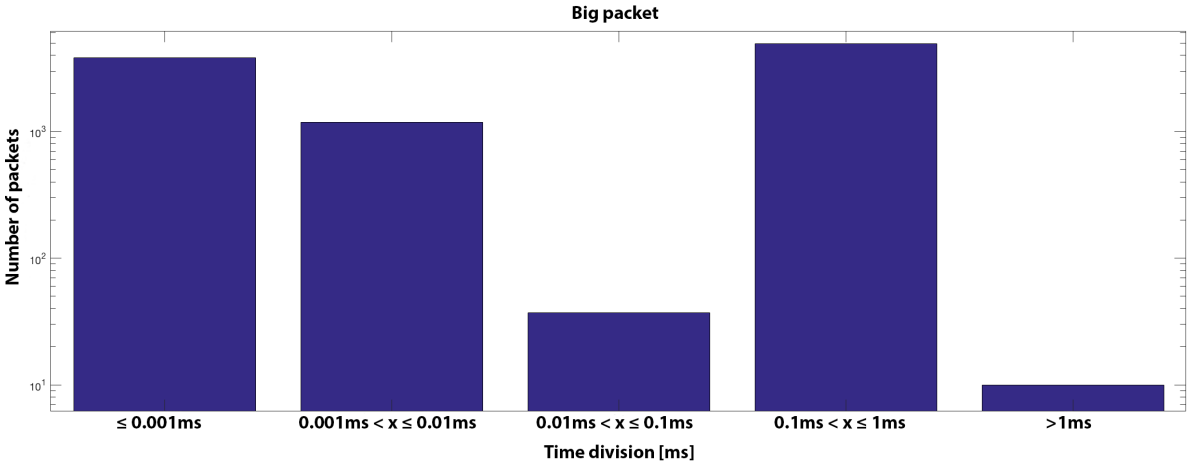


Figure 4.7: Communication of a big packet (1014bytes)

## 4.2. Communication with simulation running

The same test will now be conducted, but with a simple simulation running on the HSU. This will illustrate how much the load of a simulation influences the communication speed. The size of the packets in this test is 60 bytes.

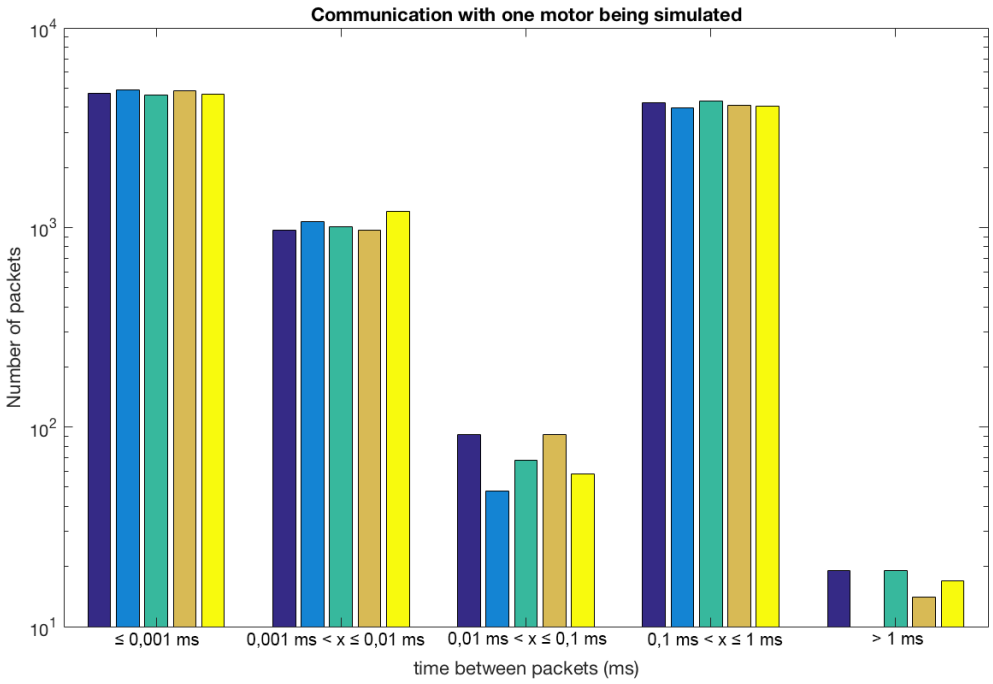


Figure 4.8: Communication with one motor

On the figure above (Figure 4.8) the test results are shown. This time 10,000 packets were transmitted again, but with the simulation simulating one motor in the background. Furthermore, this time the test was executed five times to see if there was a big deviation in the results.

Table 4.5: One Motor Running Communication Data

| | # packets | | | | |
|---|---|---|---|---|---|
| Packets > 1 ms | 19 | 10 | 19 | 14 | 17 |
| | time | | | | |
| mean | 0.0984 ms | 0.0979 ms | 0.0985 ms | 0.1009 ms | 0.0975 |
| median | 0.001 ms | 0.001 ms | 0.001ms | 0.001 ms | 0.001 ms |
| max | 10.331 ms | 4.4850 | 10.6650 ms | 14.5340 ms | 2.2710 ms |

The table above (Table 4.5) displays the data gathered from the tests. Looking at the mean values, it can be concluded that the five tests were similar. The packets that took over 1ms to be communicated are actually less than the test without one motor running in the simulation, which had 27 of them (see Table 4.3). This difference can be attributed to chance and also indicates that simulating one motor doesn't stress the system much. The mean values give a better insight in the stress one motor does result in, since it is higher than when no motors are being simulated (Table 4.4).

Another test is executed on the system, but this time under a heavy simulation strain to see if the communication still meets the requirement. The heaviest possible strain the simulation system will be put under, is when it's simulating 16 motors (see Section 1.1). On the figure below (Figure 4.9) the test results for sixteen motors are shown.

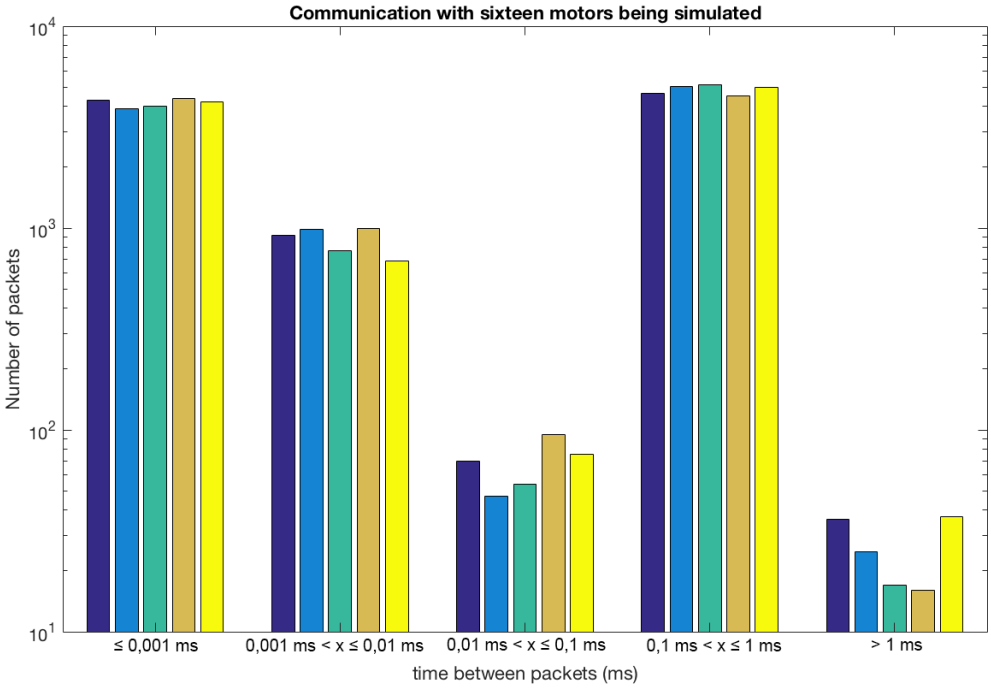


Figure 4.9: Communication with sixteen motors

The table below (Table 4.6) displays the data gathered form the tests. The averages are slightly higher as well as the amount of packets exceeding the 1ms barrier. This means that simulating more motors does indeed decrease the communication speed, but not by a very high amount. It turns out that simulating extra motors doesn't influence the communication speed much.

Table 4.6: Sixteen Motors Running Communication data

| Packets > 1ms | 36 | 25 | 17 | 16 | 37 |
|---|---|---|---|---|---|
| mean | 0.1125 ms | 0.1138 ms | 0.1126 ms | 0.1009 ms | 0.1127 |
| median | 0.001 ms | 0.1300 ms | 0.1300 ms | 0.0010 ms | 0.1240 ms |
| max | 4.3160 ms | 6.1740 | 6.4680 ms | 8.1980 ms | 6.1840 ms |

## 4.3. Communication on a 1ms tick

For the last test, the HSU was modified to send data on a 1ms tick and included the ability to increase the size of the packets being broadcasted by the HSU as the amount of motors simulated increased. As can be seen in figure 4.10, enabling the 1ms tick makes the majority of the packets take 1ms to be communicated. The small deviations from that 1ms can be explained by the switch latency. There still are packets however that still take too long to be communicated. As discussed before, this could be because of the non-realtimeness of the Linux kernel.

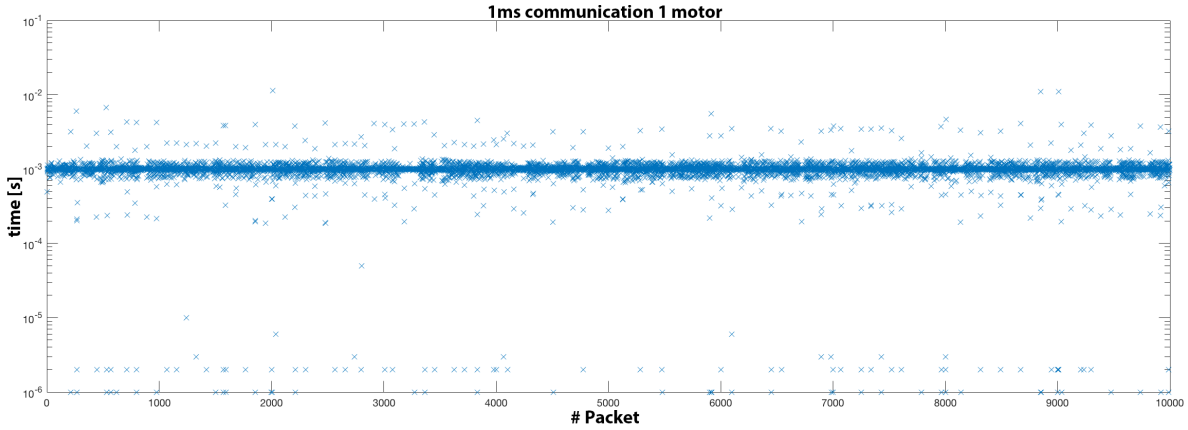


Figure 4.10: Communication with one motor on a 1ms tick

The test with one motor (Figure 4.11) is now compared with a test with 16 motors being simulated (Figure 4.12). The results were distributed in the five categories again, but now the upper limit was set to 1.02ms since the switch and test setup could delay the tick by about 0.02ms. The amount of packets for one motor that exceed the 1.02ms barrier is 2777 and 2778 for 16 motors. The average communication time of both measurements is 1ms.

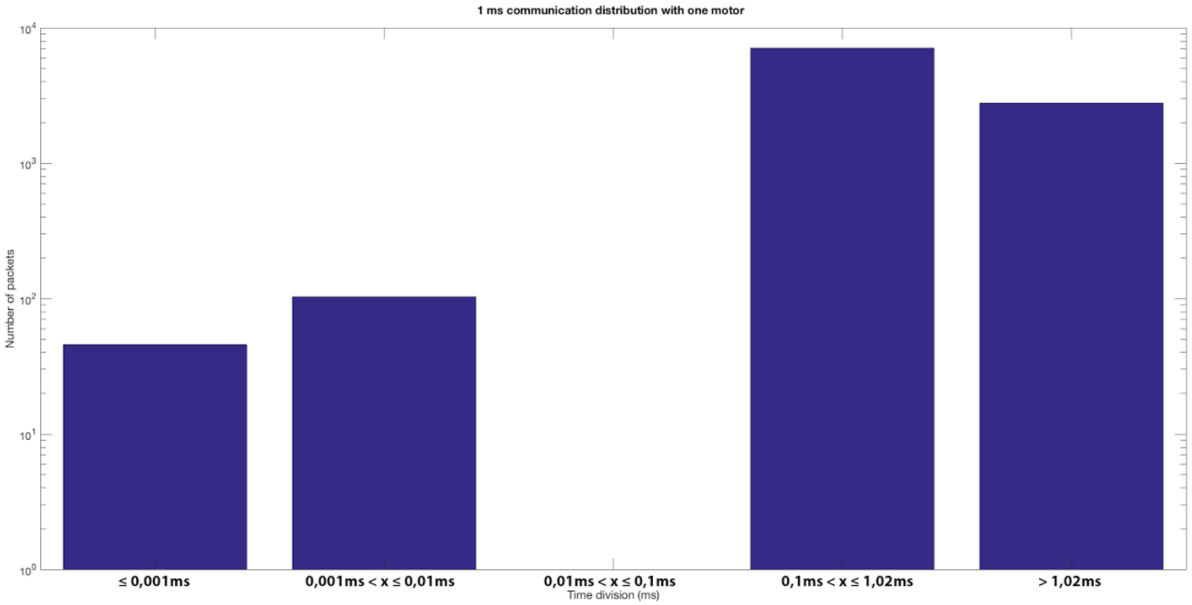


Figure 4.11: Communication distribution with one motor on a 1ms tick

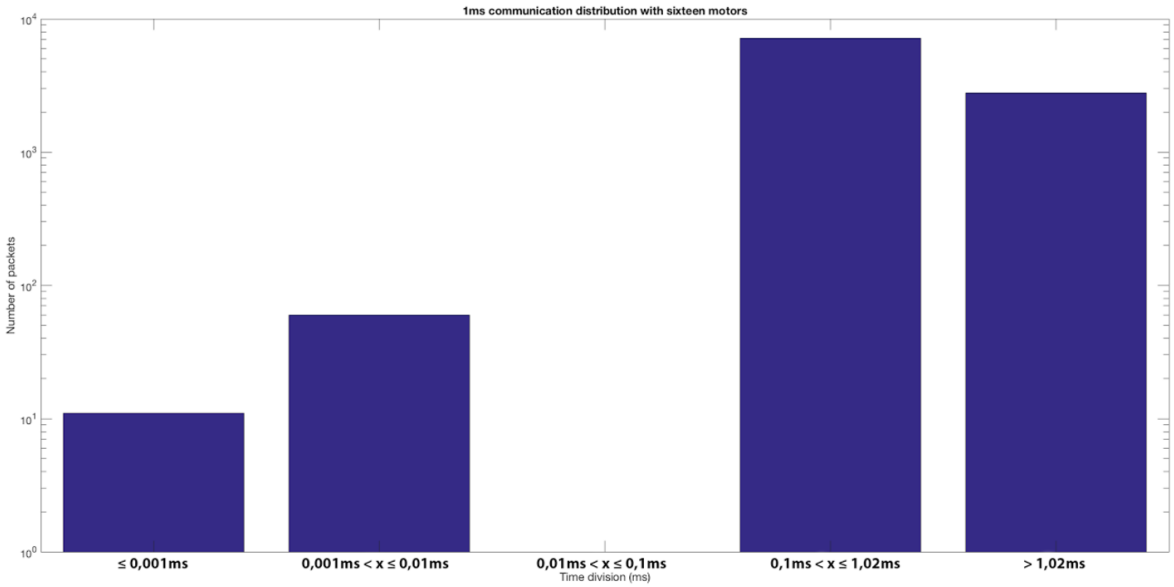


Figure 4.12: Communication distribution with sixteen motors on a 1ms tick

This data of many packages exceeding the 1ms tick is to be expected with the previous test results in mind and a possible issue being the non-realtimeness. The average communication time for 1 motor and 16 motors is 1ms however, which is promising and indicates the raw Ethernet communication could meet the 1ms requirement. To fully test this requirement however, the HSU would need to meet the 1ms tick which could be done by changing the Linux kernel to a realtime OS or baremetal C. Because with a Linux kernel it simply does not meet the requirement of a 1ms tick.

Looking closer into the deviation from the 1 ms cycle, different time ranges were analyzed. As can be seen in Table 4.7 and Table 4.8 at least 40% of the packets are communicated within 0.02ms from the 1ms tick. There is no big difference between the simulation of 1 motor and the simulation of 16 motors. If anything, the test simulating 16 motors seems to be performing slightly better than the test simulating one motor. This can be explained because the malfunctioning of the system is probably caused by non-realtimeness and the packet sizes and load on the processor by the simulation don't seem to have a very significant influence.

Table 4.7: Number of packets around 1ms with simulating 1 motor

| Range | Number of packets |
|---|---|
| 0.98 ms to 1.02ms | 4171 |
| 0.95 ms to 1.05ms | 6312 |
| 0.9 ms to 1.1 ms | 7391 |
| 0.8 ms to 1.2 ms | 8723 |
| t < 0.1 ms | 151 |
| t > 2 ms | 92 |

Table 4.8: Number of packets around 1ms with simulating 16 motors

| Range | Number of packets |
|---|---|
| 0.98 ms to 1.02ms | 4363 |
| 0.95 ms to 1.05ms | 6455 |
| 0.9 ms to 1.1 ms | 7611 |
| 0.8 ms to 1.2 ms | 8723 |
| t < 0.1 ms | 72 |
| t > 2 ms | 74 |

Another interesting thing to note is that some packets were communicated faster than 0.1ms. This can be explained because whenever one packet gets delayed the HSU will at some point notice that and make up for it by sending the packets that missed their tick in quick succession.

Finally, it can be concluded that in all the tests that have been executed, not a single Ethernet packet was dropped, so raw Ethernet should be reliable enough as a protocol.

# 5

# Conclusions and Recommendations

## 5.1. Conclusions

A raw Ethernet communication architecture was designed in this thesis. The design choices are described and compared to alternatives in Chapter 3. This design was conceived with modularity in mind, so it will be relatively easy to implement other components like the LVDT, servo and encoder. The modularity also allows additional data or control signals to be included later on. Furthermore, more command codes for the HSU can easily be implemented to for example have a unique command that requests just the temperature from one component.

This design was then tested in Chapter 4. From this evaluation it could be concluded that raw Ethernet communication is reliable and fast enough, but the 1ms tick requirement has not been met. A possible explanation for this could be the non-realtimeness of the Linux kernel running on the HSU. Other explanations could be that the switch has a larger latency or buffer time than expected or that the HSU has some sort of memory or buffer problem. But based on the test results, the most probable cause is thought to be the non-realtimeness of the HSU.

## 5.2. Recommendations

In addition to the current communication system, the following possible improvements could be looked into.

### 5.2.1. Realtimeness

As seen in the results in Chapter 4, not every packet will be sent within the 1 ms tick. The Linux kernel running on the HSU is not running realtime, which could cause the delays as have been seen in the tests. This problem could be solved by implementing a Linux kernel which is realtime like RTLinux[19].

It is also recommended to execute the tests executed in Chapter 4 again, but with a different switch that has a well defined latency. This could rule out the switch as the cause of not meeting the 1ms tick requirement. Another way to test this could be to send data form the HSU directly to a computer running Wireshark with no switch in between.

Lastly, the memory structures of the SAMA5 should be looked into, perhaps the problem was caused by some sort of memory problem.

### 5.2.2. Improve the test setup

Another recommendation would be to improve the test setup. Because of time constraints to improve the functionality of the HSU, the tests were conducted as described in Chapter 4 with a PC running Wireshark. A better test setup would be to have the HSU track all its outgoing and incoming packets in a register and export that register after testing.

### 5.2.3. Check sequence of packets

During evaluation,the sequence of the packets was never tracked. The order of the packets is important for the simulation. If they arrive in the wrong order, the simulation will use outdated, not relevant information. In the worst case this could lead to an unstable simulation. For testing purposes the packets could be marked and checked on their sequence.

### 5.2.4. Data Structures

Right now, the only component compatible with the protocol is the pin driver. Due to the lack of time and manpower there was no development made for the other components. The data structures as they are now, are developed for the pin driver but designed with modularity in mind. Not only the packets sent from the HSU and the packets sent from the component controller, also the other processes on the component controller. The data from the HSU contains the configurations for the pin driver and the data from the component controller contains the measurements done on the pin driver. Furthermore, the communication to the pin driver, which is now done by SPI-connections and by driving some Input-/Output pins, is only developed for the pin driver. The protocol is modular enough to add new data structures which could be used for components such as the encoder and the LVDT/Servo. The software is easily modified by adding or changing data structures to fulfill the needs for the other components.

### 5.2.5. Error handler

There's always a situation possible where errors occur. At the moment errors will crash the system. When the connection is suddenly lost or a component is sending strange values, nothing will be done by the code. But the code can foresee some errors. To prevent an unstable system, something like an error handler could be built in. When the system is detecting a lost connection it should be able to set up that connection again without resetting. This could be done by implementing a time-out to detect the lost connection. The connection could be recovered by going back to the initialisation phase and doing the initialisation all over again. Also, some specific tests could be implemented by checking for only the offset or a fixed value. New instructions could easily be added, because the byte for the instructions offers enough space to develop new instructions.

### 5.2.6. Multiple pin drivers on the controller

At this moment only one pin driver is implemented on the microcontroller (SAME70). To make efficient use of the space available on the PCB's the pin drivers are mounted, it could be desirable to have more than one pin driver per PCB. So a microcontroller has to support multiple pin drivers. The microcontroller is expected to be capable of implementing an extra pin driver as there are enough dedicated in- and output pins available and the processor power is thought to be sufficient. Only the communication has to change. Especially in the data structure since it has to define which values are assigned to which pin driver. This could be done by adding an extra byte containing this information or by using one of the as of now unused bits of the control-byte.

# Bibliography


[1] Prins, R.V. and De Smalen, T.J., "A wide range input and output driver for a hydraulic simulation system", June 2017

[2] Zacca, V.G. and Van Rijn, J.M.S., "A hydraulic "hardware in the loop" simulation system", June 2017

[3] Atmel, "SMART ARM-based MPU", SAMA5D3 Series, February 2016

[4] Atmel, "SMART ARM-based Flash MCU", SAME70, January 2016

[5] HedoN (2017). Hydraulieksimulatie (Opdrachtbeschrijving)

[6] Van Mieghem P. (2014). "Data Communications Networking." The Netherlands: Techne Press

[7] Standard for Ethernet, "IEEE-SA standard 802.3", 2015, Available [online] https://standards.ieee.org/findstds/standard/802.3-2015.html

[8] M. Elbeshti, M. Dixon and T. Koziniec, "An Evaluation of TCP and UDP Protocols Processing Required for Network Interface Design at 100 Gbps," 2011 IEEE International Conference on High Performance Computing and Communications, Banff, AB, 2011

[9] Lin, Z. (November 2013). "An inside look at industrial Ethernet communication protocols." Dallas. Texas Instruments Inc.

[10] Spurgeon, C.E. & Zimmerman, J. (2014). "Ethernet: The Definitive Guide, Second Edition:Designing and Managing Local Area Networks." USA: O'Reilly Media Inc.

[11] D. Eastlake 3rd, J. Abley, "IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters" Dyn, Inc., 2013, Available [online] https://tools.ietf.org/html/rfc7042

[12] Seifert, "The Switch Book", Wiley Computer Publishing, 2000

[13] IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008 , vol., no., pp.1-70, Aug. 29 2008, [online] Available: http://ieeexplore.ieee.org/document/4610935/

[14] Atmel Corporation, "Embedded Software Solution for Atmel Flash MCU"s, 2012 Available: [online] http://asf.atmel.com/docs/latest/

[15] Austinmarton, "recvRawEth.c" & "sendRawEth.c", 2012 [online] Available: https://gist.github.com/austinmarton/2862515, https://gist.github.com/austinmarton/1922600

[16] Kálmán, G. and Orfanus, D. (november 2013). "Measuring Latencies Over Industrial Ethernet Switches." Serbia, Belgrade

[17] TP-Link. (2015). "5/8-Port 10/100/1000MbpsDesktop Switch", TL-SG105 / TL-SG108, Available: [online] http://static.tp-link.com/res/down/doc/TL-SG105-108.pdf

[18] Nixtutor (22-04-2009), "Changing Priority on Linux Processes", [online]. Available: https://www.nixtutor.com/linux/changing-priority-on-linux-processes/

[19] N. Vun, H. F. Hor and J. W. Chao, "Real-Time Enhancements for Embedded Linux," 2008 14th IEEE International Conference on Parallel and Distributed Systems, Melbourne, VIC, 2008, pp. 737-740.

[20] Spurgeon, C.E. & Zimmerman, J. (2013). "Ethernet Switches: An Introduction to Network Design with Switches." Sebastopol, CA: O'Reilly Media Inc.

# A

# Appendix

## A.1. Pseudo-code of the protocol on the SAME70

```
if(packet received){
    call process_packet()
}

process_packet(){
    if(destination == MAC-address pindriver){
        if(source == MAC-address HSU){
            call initialisation_process()
        }
         else if(source == Broadcast-address){
            call communication_process()
        }
    }
}

initialisation_process(){
    switch(instruction_code){
        case 0x01:
            send back requested number of bytes
        case 0x02:
            read offset, set offset and send back offset
        case 0x03:
            send back offset
    }
}

communication_process(){
    if(instruction_code = 0x04){
        read values of HSU
        send instructions and values to component
        read values from pindriver
        send back values to HSU
    }
}
```