University of Twente

EEMCS / Electrical Engineering
Robotics and *Mechatronics*

Communication Component for Multiplatform Distribution of Control Algorithms

J.J. (Jan Jaap) Kempenaar

MSc Report

January 2014

Report nr. 001RAM2014
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

These days cyber-physical systems play an important part in life. Cyber-physical systems refer to systems with a physical/mechanical part, controlled by a cyber platform. Due to the increasing demand in functionality, their designs become more complex. In this thesis the focus is on the cyber part of the cyber-physical systems.

As a result of the increasing complexity in the software algorithms of the cyber system, the requirements for the computational platform increase. Embedded computers, mostly used for the control of cyber physical systems, do not satisfy the resource requirements of these complex algorithms. To deal with this, the complex algorithms are distributed among multiple platforms, each designed for a specific purpose. Embedded computers take care of simple, high-frequency, hard real-time loop control algorithms, while more advanced computers are used for the execution of the more complex, low-frequency, supervisory control algorithms.

Tools exist to aid in the development of the algorithms. There is no single tool that meets the requirements to develop all algorithms. Therefore, many tools are used. Via code-generation toolboxes, the tools produce implementations for the various algorithms that can be deployed on different platforms. Several deployment tools exist for this purpose.

The problem is that the current deployment tools do not offer a way to connect the algorithm applications on the various platforms to communicate with each other. This causes developers to not use the tools, but come up with their own solutions. This does not always result in optimal deployment situations.

A *Communication Component* is designed and implemented in this thesis to allow the complex low frequency algorithms, to provide their data to the real-time control platforms. The *Communication Component* consists of a server application on each device, that connects with the server application of the *Communication Component* on other devices. For the algorithm it offers an Application Pogramming Interface (API) to connect and communicate with the other algorithms via the *Communication Component*.

A platform analysis and use case analysis are performed to draft requirements for the *Communication Component*. Based on these requirements a design is made and implemented.

Ethernet is used for the network communication. Using Inter Process Communication (IPC), algorithms on different platforms are capable of connecting to the *Communication Component*.

The use case, used for the requirement analysis, is used as a basis, for a demonstrator experiment to verify the working of the *Communication Component*. The Gumstix Overo Fire is used as the computing platform in the experiment. This computing platform is currently developed as the standard embedded computing platform for setups within the Robotics and Mechatronics (RaM) group. From the results of the experiment it can be concluded that the *Communication Component* is capable of connecting the various platforms and provide a means for complex algorithms to send their data to the real-time control platform.

The Gumstix Overo Fire is not yet supported by the 20-sim 4C deployment tool. To ease the deployment process, it is recommended that platform support for the Gumstix Overo Fire is added to 20-sim 4C. Other software frameworks like OROCOS provide building blocks for developing complex sequence and supervisory control algorithms. Adding integration capability of these frameworks with the *Communication Component* is also recommended as it can aid in the development process.

Samenvatting

Cyber-fysische systemen spelen een belangrijke rol in het dagelijks leven. Cyber-fysisch systeem refereert naar systemen die bestaan uit een fysieke component, welke bestuurd wordt door een cyberplatform. Door hogere functionele eisen, wordt het ontwerp van deze systemen steeds complexer. In deze thesis ligt de focus op het cyber deel van deze systemen.

Doordat de complexiteit van de software-algoritmes toeneemt, nemen ook de eisen voor het computerplatform toe. Embedded computers, welke vaak gebruikt worden voor het besturen van cyber-fysische systemen, kunnen niet langer voldoen aan deze eisen. Om dit probleem op te lossen, worden de algoritmes verdeeld over verschillende computer platformen, elk met hun specifieke taak. Embedded computers worden gebruikt voor simpele, hoogfrequente, hard real-time regel-algoritmes, terwijl geavanceerde computers gebruikt worden voor het uitvoeren van de complexe, laagfrequente, supervisory regel-algoritmes.

Om te helpen in de ontwikkeling van deze complexe algoritmes, bestaan er verschillende tools. Er is echter niet een tool die voldoet, om al deze algoritmes te ontwerpen, dit heeft tot gevolg dat per project meerdere tools gebruikt worden. Via codegeneratie worden implementaties voor de algoritmes geproduceerd. Deze kunnen vervolgens op de verschillende computerplatformen worden uitgerold. Er bestaan verschillende tools die voor dit doel gebruikt kunnen worden.

Het probleem van deze tools is dat zij niet een manier bieden om de algoritmes van de verschillende tools op de computerplatformen met elkaar te verbinden. Het gevolg hiervan is dat ontwikkelaars een eigen oplossing ontwikkelen. Dit leidt echter niet altijd tot een optimale distributie van de algoritmes.

In deze thesis is een *Communicatie Component* ontworpen die het mogelijk maakt om laagfrequente algoritmes via een netwerk te verbinden met hoogfrequente algoritmes op andere platformen. De *Communicatie Component* bestaat uit een *Communicatie Server* die het netwerk verkeer regelt en een *Communicatie Interface* die door de algoritmes gebruikt kan worden om met de *Communicatie Component* te verbinden. Een platform en use-case analyse zijn uitgevoerd om eisen voor het component op te stellen.

Voor de netwerk communicatie is Ethernet gebruikt. Om applicaties te kunnen verbinden met de *Communicatie Component* is gebruik gemaakt van interproces communicatie.

De use-case uit de analyse is gebruikt als basis voor een demonstratie, om de werking van de *Communicatie Component* te verifiëren. Bij het experiment is de Gumstix Overo Fire als computer platform gebruikt. Dit platform is gekozen, omdat deze wordt ontwikkeld als het standaard computer platform voor de experimenteer opstellingen in het RaM laboratorium. Uit de resultaten blijkt dat de *Communicatie Component* geschikt is voor het verbinden van algoritmen op verschillende platformen via een netwerk.

De Gumstix Overo Fire wordt nog niet door de 20-sim 4C tool ondersteund. Het is aanbevolen om ondersteuning voor dit platform te ontwikkelen, aangezien deze too bijdraagt bij het uitrolproces. Andere frameworks, zoals OROCOS, bieden bouwstenen voor het ontwikkelen van complexe regel algoritmen. Integratie van deze frameworks in de *Communicatie Component* is ook aanbevolen, aangezien deze frameworks kunnen bijdragen in het ontwikkelproces.

Contents

1 Introduction

1.1 Context

These days cyber-physical systems play an important part in life, people make use of such systems in daily situations without even noticing them. Cyber-physical systems refer to systems with a physical/mechanical part, controlled by a cyber platform. Mostly, this cyber platform is an embedded computing platform. Due to the increasing demand in functionality, cyber-physical systems become more complex. This results in complex mechanical designs and complex software algorithms to control the system. One can think of image processing algorithms for position feedback, rather than using only angle or velocity values as used in the past. The focus in this thesis is on the cyber part of the cyber-physical systems.

As a result of the increased complexity of the controlling algorithms, the requirements for the computational platform used to control the physical system increase. Complex algorithms require more resources. Embedded computers are not sufficient to provide these resources and do not fulfil the requirements. To deal with this, algorithms are distributed among multiple platforms, each designed for a specific purpose. Embedded computers take care of hard real-time loop control. These are mostly simple algorithms that require few resources, but run at a high loop frequency. More advanced computers are used to execute complex algorithms. Mostly these algorithms run at low loop frequencies. For example, image processing is limited to the speed at which camera images are provided, for normal cameras this is 30 Hz.

To aid in the development of these complex algorithms, different tools and design approaches exist. Each of these tools has their specific area of expertise. Because there is no tool that fulfils all requirements for development, several tools are used in the design process. A model-driven design approach is considered a best practice in the development of these algorithms. Via code-generation toolboxes, the models of these algorithms can be implemented and deployed on the computer platforms. Various deployment tools exist for this purpose. They map the I/O of the hardware components to the algorithms.

The problem with the current tooling is that they result in several applications that run on different platforms. These applications however, do not have a way to exchange data between them. This causes developers to divert from using the available tooling and create their own solutions to deploy these various applications on different platforms. As a result, not always the ideal deployment is chosen.

1.2 Goals and Approach

The problem with the tooling and platform distribution, also applies at the Robotics and Mechatronics (RaM) laboratory. Experimental setups get more complex and incorporate more complex software algorithms. To solve the problem of connecting the different computing platforms, a *Communication Component* is designed in this thesis. The focus for the *Communication Component* is to support the computing platforms and the tools used in the RaM laboratory.

The goals of this research assignment are:

- Design and implement a *Communication Component* that allows different platforms to communicate data via a network.
- Provide a means for algorithms running on the different platforms to exchange data via the *Communication Component*.

- Demonstrate the working of the *Communication Component* by means of a demonstrator.

To achieve these goals, first a platform analysis is conducted. To analysis is used to identify the platforms and tools currently used in the laboratory. Also the way of working with the tools and the platforms is analysed. Based on a generic use case, requirements are drafted. These requirements form a basis for the design and implementation of the *Communication Component.* The use case is then used as a demonstrator in order to verify the working of the *Communication Component.*

1.3 Outline

First, background on the model-driven driven design approach and way of working, a generic embedded control software structure and the LUNA framework is provided in Chapter 2. In Chapter 3, the analysis of the different setups is discussed. A use case analysis is presented in Chapter 4. Chapter 5, elaborates on the design and implementation of the *Communication Component.* The *Communication Component* design is evaluated in Chapter 6. Last, Chapter 7 ends with conclusions and recommendations.

2 Background

2.1 Design Methodology

At the RaM group, embedded control software is developed using a model-driven design approach. The model-driven design approach is considered a best practice way of working and is detailed in Bezemer (2013). It is shown in Figure 2.1. The thesis focuses on track (b) and (c) in the approach.

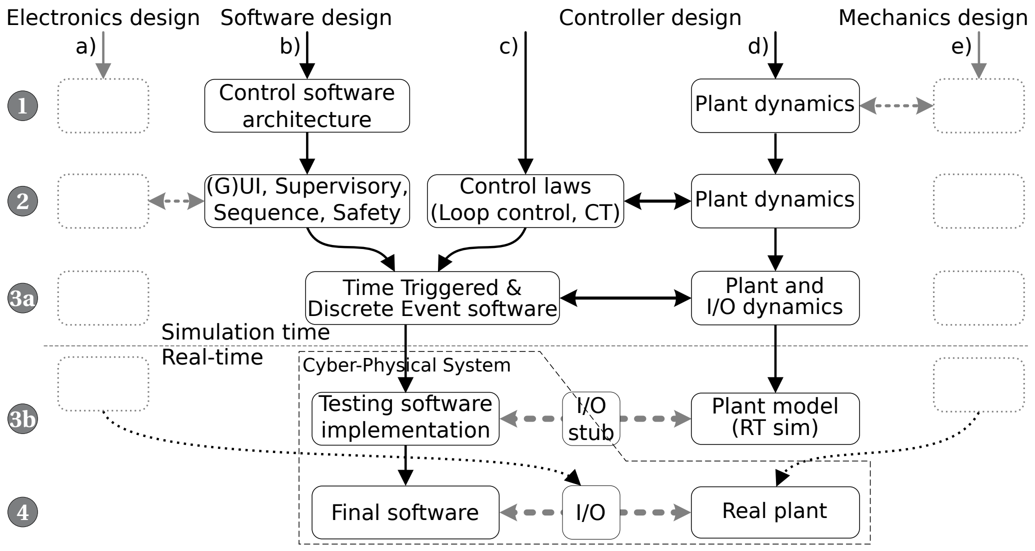


Figure 2.1: Model driven design approach.

The approach consists of four steps:

1. *Software design.* The structure of the software is designed in this step.
2. *Algorithm design.* Both control algorithms and complex algorithms, like image processing, are designed in this step. Dividing algorithms in resource-intensive algorithms and (hard) real-time algorithms is also part of this step.
3. *Verification and implementation*:
   (a) Via simulation of models, the design can be verified. Inconsistencies and other issues can be identified, without using an actual physical system.
   (b) Software is deployed on the different computing platforms and tested with simulation models of the physical system. This way, it can be verified that the software still fulfils the requirements.
4. *Realisation.* In this step, the code is coupled with hardware drivers and connected to the actual physical system.

During each step in the design approach, verification is done by means of testing and simulation. This way inconsistencies and faults can be detected and solved in an early stage. Models are used in each step for the simulation. Using models rather than the actual physical system also reduces the chance to damage the system in the design process. Models also serve a second purpose. They provide a means for developers of different disciplines to elaborate on the the design.

During development, it is important that hardware-specific implementations are kept out of the models. This allows for better reusability of the models and also allows to go back a step in the design process, should this be required.

2.2 Embedded Control Software Structure

For the development of Embedded Control Software a layered software pattern, inspired by Bennett S. (1988), is used within the RaM group, see Figure 2.2 (Bezemer, 2013).

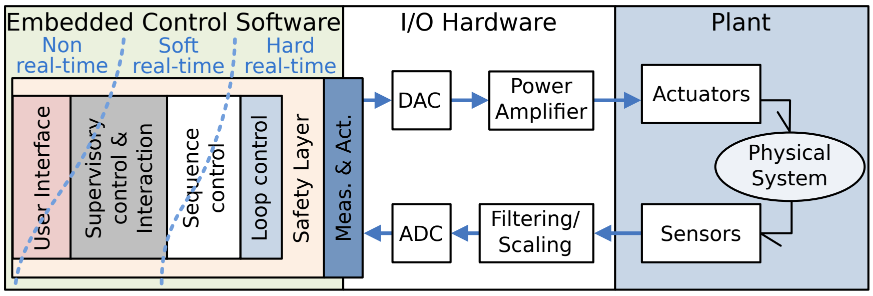


Figure 2.2: Layered embedded control software architecture

- The *Loop Control* is responsible for the control of the hardware. The *Loop Control* implementation is hard real-time, missing deadlines in this layer of the software can cause catastrophic results. For example the system can cause damage to itself or its surroundings.
- The *Sequence Control* is responsible for controlling the *Loop Controllers*. The *Loop Controllers* perform simple tasks and have no knowledge of other parts of the system. The *Sequence Controller* does have this overview and provides the setpoints for the various *Loop Controllers*. Based on the application, the *Sequence Control* is either hard or soft real-time.
- The *Supervisory Control & Interaction* layer contains complex algorithms. For example path planning algorithms are in this layer, calculating paths for the robotic device which are then passed on to and executed by the *Sequence Controllers*. But also image processing algorithms that are used for position feedback and planning belong to this layer. Algorithms in this layer do not cause harm when deadlines are missed, therefore algorithms in this layer are considered soft or non real-time.
- The *User Interface* is non or soft real-time. When the application is not able to present new updates in the user interface in time, it does not cause catastrophic events. It is however desirable to have some responsive behaviour in this layer of the software.
- The *Measurement & Actuation* is responsible for interfacing with the hardware. Software drivers to control the hardware, or to read values from sensors of the system are part of this software layer. Also signal filtering and scaling is part of this layer. Disturbances in the input signal can have a negative effect on system stability. Scaling is used to convert the input signals to values that can be understood by the control algorithms.
- The *Safety Layer* surrounds all the layers. It verifies that incoming sensor signals or outgoing control signals are within the limits set for a particular system. It also checks whether algorithms in the other control layers do not perform unintended behaviour. For example a state machine that tries to continue to a state that is not allowed.

2.3 LUNA Framework

The LUNA Universal Networking Architecture (LUNA) framework is developed within the RaM group (Wilterdink, 2011). It is designed as a CSP-capable hard real-time framework that can be used in code generation for graphical CSP models. The architecture of the LUNA framework is shown in Figure 2.3.

The design of the framework is component based. Depending on the capabilities of the deployment platform and the requirements of the application, different components can be enabled or disabled. The CSP capabilities of the framework are decoupled from the other components of the framework and added as a single component. Therefore, CSP can be disabled. This allows the framework to be used in non-CSP based applications as well.

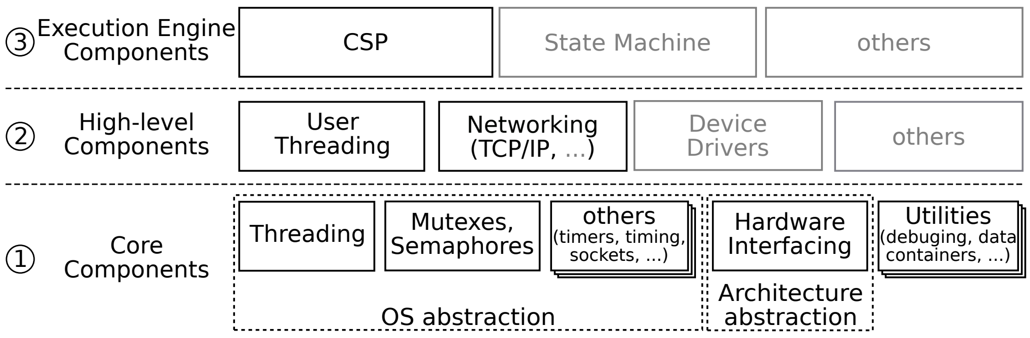


Figure 2.3: LUNA framework architecture overview.

A requirement in the development of the framework is platform independency. To achieve this an abstraction layer is incorporated into the design of the framework. This abstraction layer has been split into an Operating System (OS) and Architecture abstraction layer (see Figure 2.3). Applications can utilise this feature to improve portability between different platforms.

3 Platform Analysis

3.1 Introduction

Using a model-driven design approach is considered a best practice way of working in designing embedded control software for a mechatronic system (Bezemer, 2013). Keeping models abstract by not including implementation specific code and using replacement tokens, model quality is improved and more suitable for reuse. Broenink et al. (2010) illustrate this with the example in which 20-sim models are used for controller design, and gCSP models for software design.

In order to keep focus only on controller design, Controllab Products offers a tool-chain in which focus is on controller design and not on software design. The tool-chain consists of 20-sim for design and implementation, 20-sim 4C is used in deployment (Controllab, 2013). 20-sim (4C) is not the only model-based toolchain available, Simulink/Matlab is also a model based design and analysis tool which is used in control engineering (Mathworks, 2013). For deployment either a Personal Computer (PC) with the Windows OS can be used or a dedicated target running xPC real-time kernel (Mathworks, 2013). The only objection against Simulink/-Matlab is that it is restricted to vendor-specific tools and targets and do not have a strict separation between the development steps (Broenink et al., 2010). Table 3.1 gives an overview of these two tool-chains and the way of working with these tool-chains.

Design	Implementation	Deployment	Monitoring and Control
20-sim	20-sim	20-sim 4C	20-sim 4C
Simulink	Simulink	Simulink (WRT[1]) / Simulink (xPC[2])	Simulink

Table 3.1: Tool-chain overview and their way of working.

In the setup analysis, the current way of working with the setups and the available tools is considered and compared to the described "ideal" way of working in Table 3.1. The setups chosen for the analysis are used in currently active research projects. They also represent the different fields of research in which the RaM group is active. The setups included are:

- Variable Stiffness Actuator (VSA) UTII
- Bipedal Walker
- Parallel bars setup
- Microrobotic setup
- Flexible needle setup

A more detailed explanation of the setups is given in Appendix A. The analysis is based on conversations with the developers of and researchers working with the setups and consulting the manuals in theses (Ketelaar, 2012; Geus, 2012).

[1]Windows Real-Time Target
[2]xPC real-time target

3.2 Setup Analysis

3.2.1 Way of Working

In the way of working with the various setups, different tools are used in the design, implementation and deployment steps. An overview of the way of working with the setups and how tools are used in this way of working is given in Table 3.2, a more detailed analysis can be found in Appendix A. There are several reasons why a certain tool or tool-chain is chosen for usage with a setup. These reasons mostly include the familiarity of the developer with the tool, cost of the tool/tool-chain or the speed at which it produces a proof of principle.

Setup	Design	Implementation	Deployment	Monitoring/ Logging
Bipedal Walker	20-sim	Simulink	Simulink (WRT[3])	Simulink
VSA UTII	20-sim	Simulink	Simulink (WRT)	Simulink
Parallel bars	20-sim	20-sim	20-sim 4C	20-sim 4C
Microrobotic setup	Simulink	Simulink	Simulink (xPC[4])	Simulink
Flexible needle setup	20-sim/ Simulink	Custom framework	Custom framework	via Custom framework

Table 3.2: Way of working for the experimental setups.

The Parallel bars setup and the Microrobotic setup both use a tool-chain provided by one vendor. These are respectively the 20-sim/20-sim 4C tool-chain and the Matlab/Simulink with xPC target tool-chain. For these setups, additional effort was put in setting up the setup with the tool-chain. According to the developers, this initially took more time to create the setup, but in the long term made it easier to quickly conduct experiments on the setup as the tool-chains decrease the time to implement new designs. Also no effort has to be put in manual conversion of models from one tool to the next tool.

The Bipedal Walker and the VSA UTII setups use multiple tools in their way of working. 20-sim is used for the initial design and simulation of the controller, Simulink is then used for deployment using the Windows Real-Time target. This is less convenient as two different tools are used within the design and deployment process, it requires the developer to be proficient in multiple tools. In addition, between design and deployment, a conversion step has to take place in order to conduct an experiment. The conversion is mostly done manually, because no conversion tools exist yet. The conversion process is therefore prone to errors. From an economical point of view this is also less ideal as licences for multiple software tools need to be acquired for all tools.

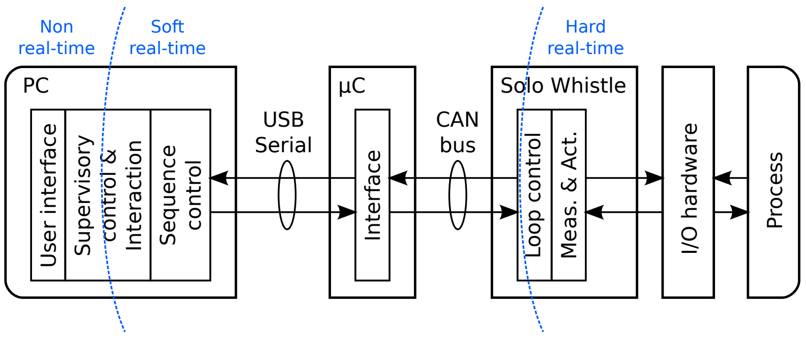
The Flexible needle setup, only uses design tools for the controller design. A custom framework is developed for the deployment. The framework allows for reusable code in the deployment application and minimises deployment time. It also controls functions for standard procedures like the homing procedure. However including new controller designs is a manual process that is prone to errors.

3.2.2 Hardware Components

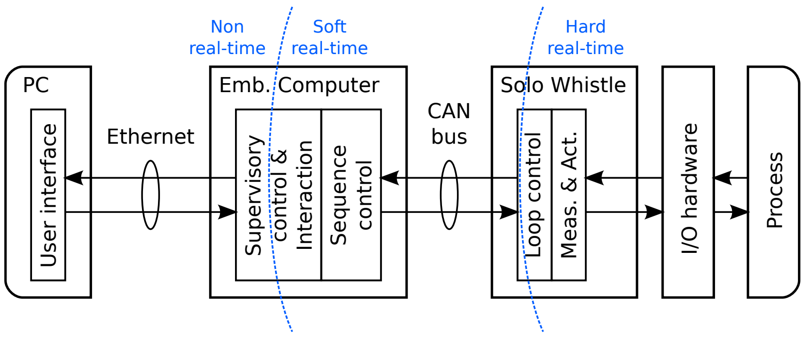
In the setups both embedded computers and PCs are used for different purposes. Another common hardware component in the setups is the *Solo Whistle* motor controller from ELMO Motion Control (Elmo Motion Control, 2013). When the setups are compared to the generic embedded control software structure from Bezemer (2013), detailed in Section 2.2, two differ-

[3]Windows Real-Time Target
[4]xPC real-time target

ent mappings can be identified in the setups using tools for deployment, these are shown in Figure 3.1.



(a) Deployment architecture with embedded controller (μC) as hardware interface.



(b) Deployment architecture with embedded computer as deployment target.

Figure 3.1: Software to hardware deployment architectures identified in the research setups.

Both mappings in Figure 3.1 use the *Solo Whistle* to execute the hard real-time loop control. When looking at higher-order control algorithms, such as the sequence and supervisory control, there are differences. Different platforms are used for the execution of these control loops.

Using Simulink Windows Real-Time Target as the deployment tool, results in the mapping shown in Figure 3.1a. The Simulink models are run on a PC with a Windows OS. The OS on the PC is not designed for performing real-time tasks, executing the sequence control and supervisory control on this platform is therefore not recommended as it is not guaranteed that deadlines are met, for example other applications can cause timing issues if they take too much CPU time. If the PC is fast enough, it will work. The microcontroller, depicted with μC, is used as an interface between the PC and the hardware. This is due to the limited I/O on the PC.

The Simulink xPC target deployment tool and the 20-sim 4C deployment tool use a dedicated target to deploy their control algorithms, this is shown in Figure 3.1b. The sequence and supervisory control loops run on a dedicated real-time platform. The PC with the monitoring and control is separate from this real-time platform. The real-time platform makes sure that adequate resources are available for the control loops in order to meet their deadlines.

The approach with the dedicated real-time platform can be considered the better of the two mappings. It has a clear separation between the real-time and non real-time tasks and also separates these per platform. Additionally the real-time control loops are executed on a dedicated real-time platform while with Simulink Windows Real-Time Target, the real-time control loops are run on a non real-time platform.

3.2.3 Observations

In the two medical setups, the Flexible Needle Setup and the Microbotic Setup, the algorithms are not limited anymore to dynamic equations. Feedback is not only provided by sensors but also by image processing algorithms. Because image processing is part of the control loop, this creates new requirements for the development and deployment tools. Not all tools support toolboxes with these algorithms yet. Also for the Flexible Needle Setup, a pathplanner algorithm is provided by a third party. Due to the platform requirements of this algorithm, deployment of this pathplanner is limited. These new requirements lead to certain choices when creating an experimental setup and can therefore also cause the choice of not using a tool for deployment.

3.3 Conclusion

Several tools and tool-chains are available to aid in the design and deployment of controllers for embedded systems. However the tools do not always provide the required functionality for a specific setup. 20-sim for example does not include image processing toolboxes which are required by the medical setups. This leads to the usage of multiple tools, or to not use tools at all.

Not using tools in the design workflow is not recommended. Therefore, a way needs to be provided such that these tools still can be used.

4 Communication Component Analysis

4.1 Use Case

An observation from the analysis of previous chapter is that more than just dynamic equations is part of the control loops, also algorithms like image processing are part of the control loops. However these algorithms limit the use of the current tools as they do not have the toolboxes to support these algorithms. In order to still add the algorithms within the control loops but not limit the use of the tools, there should be an easy way to integrate these with the tools. Such functionality can be provided by a *Communication Component*. A generic overview of this is shown in Figure 4.1.

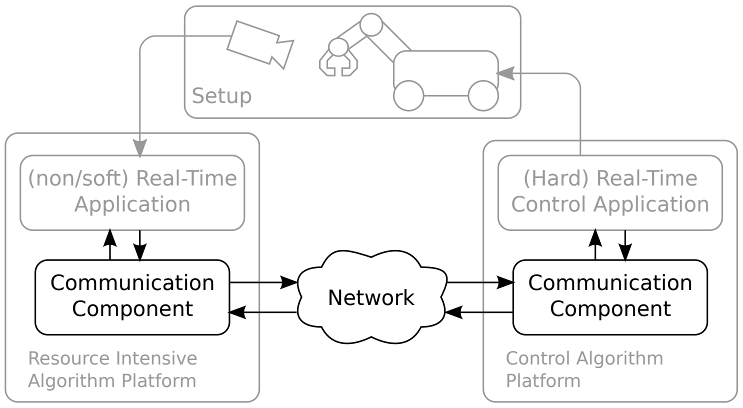

Setup
(non/soft) Real-Time Application
(Hard) Real-Time Control Application
Communication Component
Network
Communication Component
Resource Intensive Algorithm Platform
Control Algorithm Platform

Figure 4.1: Abstract overview of a Communication Component within an experimental setup on multiple platforms.

The setup of Figure 4.1 consists of three parts:

- The mechanical *Setup*.
- The *Resource-Intensive Algorithm Platform*.
  On this platform resource-intensive algorithms, like image processing are executed. These algorithms generally run at a much lower frequency than the control loops on the *Control Algorithm Platform*. Also missing a deadline in these algorithms is not as strict as on the *Control Algorithm Platform*, the algorithms are soft or non real-time.
- The *Control Algorithm Platform*.
  The real-time control loops are executed on this platform. Mostly this platform is an embedded device with less resources available as the *Resource-Intensive Algorithm Platform*. Algorithms on this platform can for example be the controller models designed in the 20-sim tool.

The two computing platforms are different platforms due to their hardware requirements. Therefore in order to connect the platforms network communication is required. Depending on the type of data that is exchanged between the devices, it may be required that the network is real-time.

The *Communication Component* acts as the communication platform between the algorithms on the two platforms. This allows the control application on the *Control Algorithm Platform* to receive setpoints from algorithms on the *Resource Intensive Algorithm Platform*. The applications run on different platforms with different hardware and OSs, the *Communication Component* must be compatible with these different platforms.

If the *Control Algorithm Platform* should have adequate resources to execute both the control algorithms and the resource intensive algorithms, than the *Communication Component* can connect them on the same device, this is shown in Figure 4.2. Here the setup only consists of two parts: The mechanical *Setup* and the *Control Algorithm Platform*. Instead of a dedicated platform for both, a single platform is present. The *Communication Component* still acts as a data exchange component in order to combine the two algorithm applications.

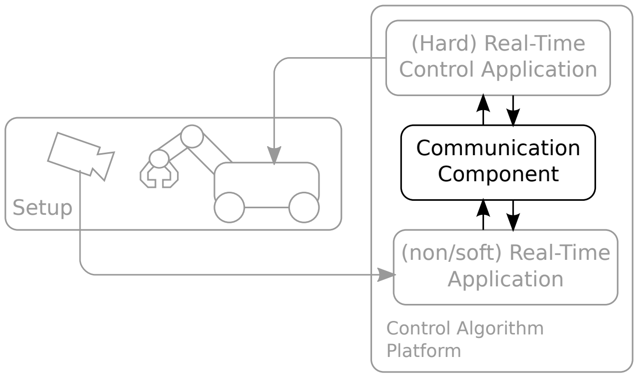

(Hard) Real-Time Control Application
Communication Component
Setup
(non/soft) Real-Time Application
Control Algorithm Platform

Figure 4.2: Abstract overview of a Communication Component on a single platform.

Because in research the focus is on the development of the algorithms and not on the integration of the algorithms, the *Communication Component* should be easy to integrate.

4.2 Requirements

The use case of previous section illustrates how a *Communication Component* could be included in a setup and shows some requirements for this component.

Two parts are considered for the *Communication Component*, these are shown in Figure 4.3:

- *Communication Interface*
  This is the interface between the algorithm application and the *Communication Component*. It defines how the application communicates with the component.

- *Communication Server*
  The server regulates the data and ensures that data is send to the correct applications that are connected to the component.

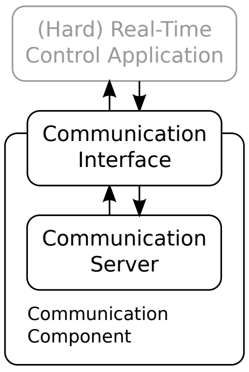


Figure 4.3: Overview of the two parts in the Communication Component.

4.2.1 Functional Requirements

General Requirement

Requirement 1: *The Communication Component must be platform independent.*

The Communication Component is used on multiple platforms, this means that it should work on those different platforms. In order to achieve this, the Communication Component should be design to be platform independent.

Communication Interface Requirements

Requirement 2: *The interface must support integration with the 20-sim 4C deployment tool.*

The 20-sim 4C deployment tool allows the user to connect the various I/O ports of the model, to the hardware outputs of the device. Connection the I/O ports of the model to the Communication component via a communication interface should follow a similar procedure. The user is already familiar with this procedure and does not need to learn this. Additionally tooling does not need to be modified if the interface can be included, similar as hardware I/O ports.

Requirement 3: *The interface must support integration with custom code.*

Some of the algorithms are provided or generated as plain source code. There should be an Application Pogramming Interface (API) available that allows the application to connect with the *Communication Component*.

Requirement 4: *The interface should include a safety layer.*

Whenever a connection is lost, or packets do not arrive at their destination, the communication interface should still be able to provide data when the application requests it.

Requirement 5: *The interface could support integration with the Simulink tool.*

Simulink and Matlab provide additional toolboxes, such as image processing, which are not part of 20-sim. Adding functionality from these tools could therefore be useful. Including these tools can have a similar approach is the integration described for 20-sim in Requirement 2.

Communication Server Requirements

Requirement 6: *The communication server must support multiple loop frequencies and real-time layers.*

In the use case of Section 4.1 multiple platforms perform their algorithms at different frequencies and in different real-time layers, described in Section 2.2 (Bezemer, 2013). The communication server must be able to cope with these different loop frequencies and real-time layers.

Requirement 7: *The communication server must be real-time capable.*

The main control loops are mostly hard/soft real-time. In order to integrate components into these loops, the communication also needs to be real-time capable.

Requirement 8: *The communication server must support network communication.*

The platforms in the use case of 4.1 are connected via a network. Therefore the communication server should support network communication in order to achieve this.

Requirement 9: *The communication server could support asynchronous communication.*

Some events do not happen at a fixed interval, for example an emergency stop button only fires once. The interface could support asynchronous communication to support such events.

Requirement 10: *The communication server could support on target communication.*

Sometimes the control platform has adequate resources to execute both resource intensive algorithms and control algorithms. This would require that those algorithms can communicate on the same platform. The communication server could also facilitate in communication between these algorithms or applications by providing communication on the same device.

4.2.2 Non-functional Requirements

General Requirements

Requirement 11: *The communication component should be lightweight.*

The processing time of the communication component should be as low as possible, such that impact on the application is kept to a minimum.

Requirement 12: *The communication component should be easy to deploy.*

Integrating the communication interface, or setting up the communication server should not take a lot of effort.

5 Design and Implementation

Based on the requirements from the previous chapter, an implementation of the *Communication Component* is made. The *Communication Component* consists of three parts:

- Inter Process Communication (IPC).
- A *Communication Server*.
- *Network Communication*.

An overview of these parts is given in Figure 5.1.

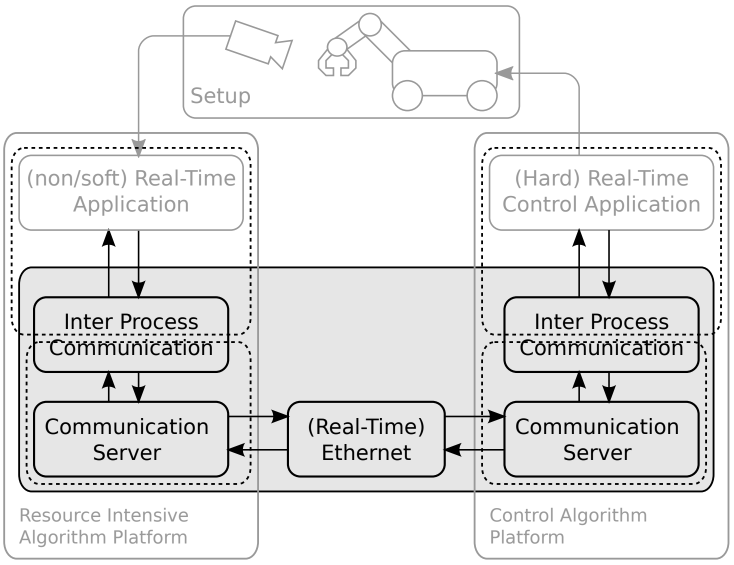

Setup
(non/soft) Real-Time Application
(Hard) Real-Time Control Application
Inter Process Communication
Inter Process Communication
Communication Server
(Real-Time) Ethernet
Communication Server
Resource Intensive Algorithm Platform
Control Algorithm Platform

Figure 5.1: Abstract system overview, for the Communication Component.

Communication needs to be provided between the application processes and the *Communication Server* process, indicated by the dashed line in Figure 5.1. IPC provides in this communication. Network communication is used to connect the different platforms.

In order to regulate the transfer of the data between applications and devices a management application is required. The communication server serves this purpose. It checks if data is available and transfers it to the correct destination.

5.1 Network Communication

Requirement 8 states that the *Communication Server* must be capable of network communication. There are several options to create a network and connect devices to them. Ethernet is chosen for implementation in the *Communication Server*. Ethernet is available on PCs and also embedded computers often have Ethernet hardware. Therefore setting up an Ethernet network does not require a lot of effort.

A downside of Ethernet is that it is not a real-time network protocol. To prevent multiple senders writing data on the same line, a mechanism for collision detection is implemented. This mechanism is called Carrier Sense Multiple Access with Collision Detection (CSMA/CD). When a collision is detected, a random time-out value is chosen, after which the sender tries again. This leads to indeterministic delays. Specially when network load is high, there is a large chance that a collision between two sending nodes can occur.

5.1.1 Real-Time Ethernet

Several protocols for usage on Ethernet exist. In industry there are several protocols used for real-time network communication, an overview of these protocols is given in Table 5.1. A more detailed explanation of the real-time Ethernet protocols is given in Appendix C.

Protocol	Open Protocol	No specific hardware	No specific drivers	Deterministic Behaviour	No dedicated network	Seperation of RT and no-RT	Overhead
Ethernet POWERLINK	+−	++	−−	++	−−	+	+−
RTnet	++	++	−−	++	−−	++	+−
EtherCAT	−	−−	−−	++	−−	+−	+
Real-Time Publisher Subscriber (RTPS) protocol	++	++	++	−	+−	−	−
UDP/IP	++	++	++	−	+−	−	+
TCP/IP	++	++	++	−−	+−	−	−−
Ethernet (Raw)	++	++	++	−	+−	−	++

Table 5.1: Comparison of Ethernet protocols. + is considered better.

In order to implement the protocols, documentation should be available. This is depicted in Table 5.1 with *Open Protocol*. Some real-time protocols require subscription to the organisationto obtain the documentation, like EtherCAT and POWERLINK. This limits the availability. POWERLINK however, has more material available for non-members so it is scored higher. RTPS and RTnet are both open protocols and documentation can be found online. UDP, TCP and Ethernet are also available via the Internet for developers, also most OSs have implemented these protocols so they are easy to use in applications.

The real-time protocols, POWERLINK, RTnet and EtherCAT, each require special drivers in order to make the Ethernet stack of the operating system deterministic. In addition EtherCAT also requires special hardware as processing of Ethernet frames is done on the hardware. This gives additional requirements for both drivers and hardware, making it less easy to set up and harder to port to multiple platforms. The upside of this is that these networks are real-time.

Real-time protocols require that all nodes that are connected on the same network, implement the same protocol. If a node is connected that does not implement the protocol, the deterministic behaviour is not guaranteed. Therefore the real-time protocols required a dedicated network. The other protocols do allow to add multiple additional nodes, however more nodes means that there is more traffic. The chance of collisions is higher when more nodes are connected in the network, therefore the amount of nodes should be limited.

For the *Communication Server*, raw Ethernet is chosen for the implementation. There is no additional overhead of protocols mapped onto the Ethernet protocol, like UDP or TCP. TCP adds additional overhead because all data packets need an acknowledgement from the receiver. Most OSs offer a means to send raw Ethernet data packets on the network. Therefore it is also easy to deploy on multiple platforms.

In LUNA, the Socket component was restructured due to the implementation of the *Communication Server*. It now provides a generic *ISocket* interface for sending and receiving data via the network. UDP/IP, TCP/IP and raw Ethernet are the available protocols in LUNA.

In order to deal with the collision problem of the Ethernet protocol, a direct connection is used. Ethernet offers duplex communication, so on a direct connection no collision can occur due to multiple senders.

5.1.2 Datapacket Protocol

In order to identify the data send via the Ethernet network, data packets need to be encoded in a format which is understandable for sending and receiving applications. For this purpose the Abstract Syntax Notation One (ASN.1) notation (Telecommunication Standardization Sector of ITU, 2002a) and encoding (Telecommunication Standardization Sector of ITU, 2002b) is used to define packet structures. The standard provides a plain text format to describe the data structure with clear encoding rules.

There are interpreters available that can convert a document with data structures described in ASN.1 format to C-code for encoding and decoding data structures. The ASN.1 compiler by Lev Walkin (Lev Walkin, 2013) is an example of such a tool that is freely available and open source. There are also closed source solutions available. These compilers have not been used because they use dynamic memory allocation in order to encode or decode the data structures. Instead encoding and decoding functions were developed, which do not use dynamic allocation so that the behaviour is more deterministic.

5.2 Inter Process Communication

There are several ways to implement IPC into applications. The LUNA framework provides IPC by means of rendezvous channels or *Shared Memory Objects*. The channels implement a subset of the *Message Queue* API, while the *Shared Memory Objects* simply provide an interface to communicate via a shared block of memory provided by the OS. Both *Message Queues* and *Shared Memory Objects* can also be used standalone in order to implement IPC. An overview of these three IPC methods is given in Table 5.2.

IPC type	Non-blocking	Low overhead	Easy integration	Cross platform
Messaging Queues	+	−	++	+−
LUNA IPCRendezvousChannel	−	−	++	++
Shared Memory Objects	++	++	+−	+−

Table 5.2: Requirements overview for IPC

5.2.1 Message Queues

Message Queues are buffers that hold messages. OSs offer these as a means to communicate between processes on the same system. A standard API is provided by the POSIX standard (The Open Group, 2013). OSs implementing this standard all use the same function interface in order to send and receive data from the queues. This allows for easy porting of applications as code does not need to be changed and the application only has to be recompiled for the particular platform.

In order to send data to other applications a *Message Queue* is registered with the operating system, a process can then write information (messages) to the queue. The OS takes care of the management of the queue.

There are two approaches to read information from the queue. The first approach is a call to a read function that blocks until data is read from the queue or an error occurs. This approach blocks thread until data is received, therefore care should be taken using this approach. The other approach is based on interrupts. When the queue receives data, it sends an interrupt. Applications can register to this interrupt and provide a callback function. This approach does not block execution of the application but can preempt important tasks. Because preemption can take place in important tasks, care should also be taken with this approach.

5.2.2 LUNA IPCRendezvousChannel

LUNA provides an IPC implementation by means of channels. For the implementation of these channels, LUNA uses the *Message Queues* provided by the OS of the POSIX standard. This means that the channels have the same limitations as the queues. For obtaining data from a channel a read function is used, which blocks until data has been received. When programming applications, this should be taken into account.

The advantage of using LUNA IPCRendezvousChannels over *Message Queues*, is that LUNA provides an abstraction layer to interact with the channels (Section 2.3). The abstraction layer increases portability of the application because the application only needs to be recompiled with a LUNA library for the specific platform.

5.2.3 Shared Memory Object

Shared Memory Objects are another means of providing communication between processes on the same OS. In this case two or more processes each share the same memory object in the OS. A POSIX API is defined to create *Shared Memory Objects*. In order to read or write data, processes can simply read or write to and from the *Shared Memory Object* as if it is part of their own application memory. However there is no read/write regulation at all, so multiple processes can access the same part of the memory at the same time, resulting in undefined behaviour.

In order to deal with this, mechanisms need to be implemented which regulate the usage of the *Shared Memory Object*. For example semaphores/mutexes can be used to only allow one process to access the memory object at the same time. This gives the developer a lot of freedom in the use of the *Shared Memory Object*, but can cause significant issues and overhead when done wrong.

The LUNA framework only provides an OS abstraction layer to create a *Shared Memory Object* and to read and write data in the *Shared Memory Object*. It does not include mechanisms, like mutexes, to protect that data from multiple readers or writers.

5.2.4 Shared Memory Object with Lock-Free Queue Implementation

A *Shared Memory Object* is used for the IPC with the *Communication Component*. The *Shared Memory Object* allows for a lock-free data read and write implementation. This way, processes and applications do not block when reading and writing data to other processes and applications.

In order to regulate the memory access, a lock-free queue algorithm (Michael and Scott, 1996) is implemented in the *Shared Memory Object*. A lock-free queue implementation already exists in the current version of the LUNA framework (Wilterdink, 2011), this implementation is used as the basis for the lock-free queue in the *Shared Memory Object*.

The current implementation of the lock-free queue however, does not suffice for implementation with a *Shared Memory Object*. It uses pointers to manage the location of the nodes in memory. This is a problem because the base address of the *Shared Memory Object* is different per application. The result of this is that a pointer's address-value points to completely different memory per application.

Another problem with the current implementation of the lock-free queue, is that the value for each node is not stored within a queue node. Instead a pointer is set to the address where the value is stored. The consequence of this is that the value can be stored in a memory location that is outside the *Shared Memory Object*.

To solve these problems, the lock-free queue was extended with memory management based on array indices and *Queue Nodes* that store the value within the *Shared Memory Object*. The structure of the *Shared Lock-Free Queue* is shown in Figure 5.2.

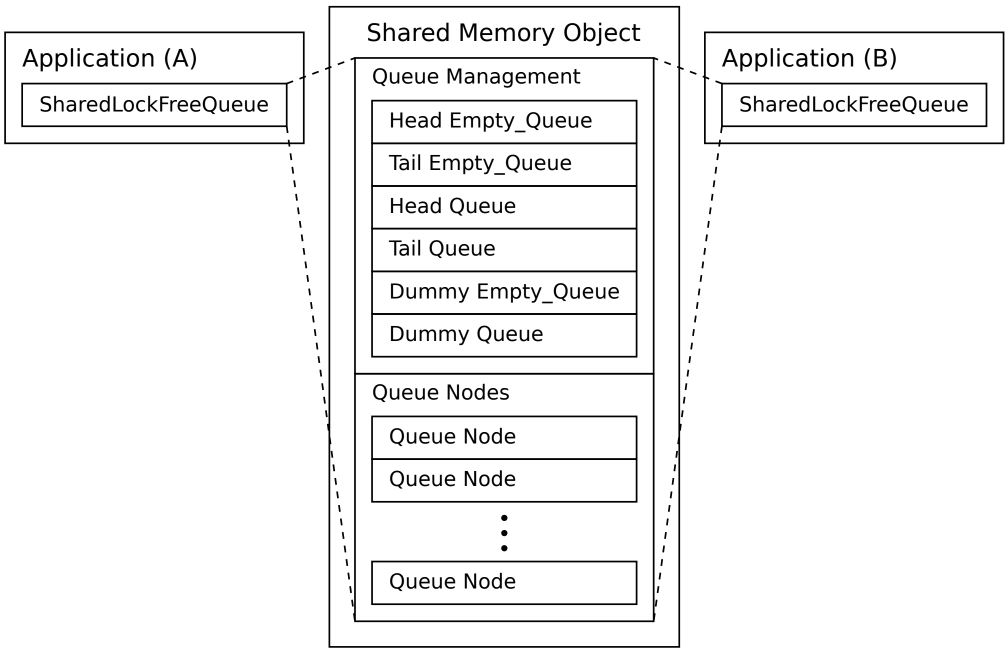


Figure 5.2: Memory structure of the Shared Lock-Free Queue.

The queue is only used to transfer basic datatypes, such as integers and floating points, to other applications. The queue is also limited to one datatype per queue. This means that the size of each node is fixed. The nodes can be mapped into the *Shared Memory Object* as an array of *Queue Nodes*. Array indices are used for the location management of the *Queue Nodes*, instead of pointers.

The *Head* and *Tail* management objects are also moved to the *Shared Memory Object*, so each application can access these. When an application connects to a *Shared Memory Object* with lock-free queue, pointers are set to the correct base address of the *Queue Management* and *Queue Nodes*.

The implementation of the *Shared Lock-Free Queue* is added to the *LUNA* framework in the *Utility* component as the *SharedLockFreeQueue* class.

5.3 Communication Server

5.3.1 Software Structure and Implementation

The software for the *Communication Component* is divided into three packages, as shown in figure Figure 5.3:

- *Communication Package*
  Classes and functions for network communication
- *API Package*
  Classes and functions for communication with the server via IPC from other software.
- *CommunicationServer Package*
  Classes and functions to regulate and process data from the applications and the network.

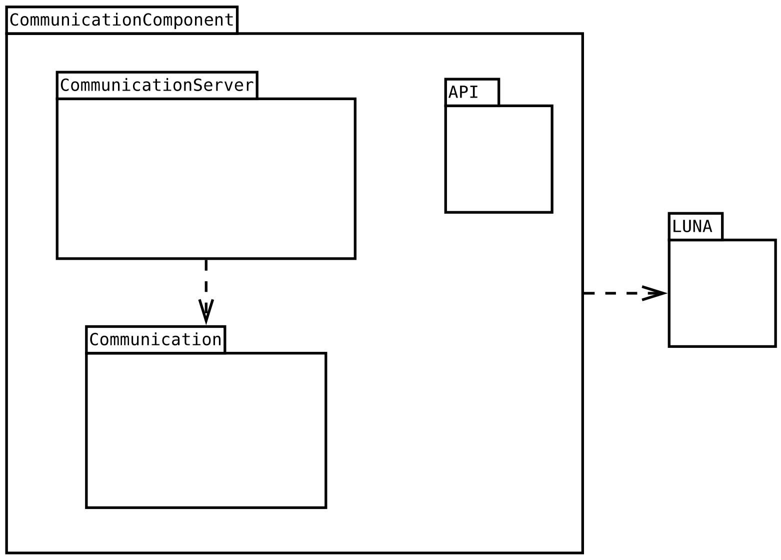


Figure 5.3: Package structure overview of the Communication Component software.

Requirement 1 states that the software of the *Communication Component* must be platform independent. To achieve this the LUNA framework is used (Section 2.3). The dependency on the LUNA framework is also shown in Figure 5.3.

In the source code, the *Communication Component* is referred to as the *Communication Interface* due to naming convention at the time of writing the code.

Communication Package

In the *Communication Package* two interfaces are defined for the network communication. For supervision of the network protocol a generic implementation is part of the *Communication Package* in to form of the *RaMSupervisor.*

- *NetworkSender Interface*
  Defines a generic interface to send data to a specified target.
- *NetworkReceiver Interface*
  Defines a generic interface to receive data.
- *RaMSupervisor*
  Generic implementation for regulation of data that is received or send in the RaM data protocol format, described in Section 5.1.2.

An Ethernet implementation is made for the sender and receiver interface in the *Communication Package.* The generic supervisor implementation is extended with an Ethernet implementation. Encoding and decoding functions for the data packet protocol, defined in Section 5.1.2, are grouped together in the *RaMProtocol* class. Implementing a different network communication protocol, only the Ethernet classes need to be replaced with the implementation of the new communication protocol. For example, the Ethernet classes could be replaced with an implementation to support serial communication.

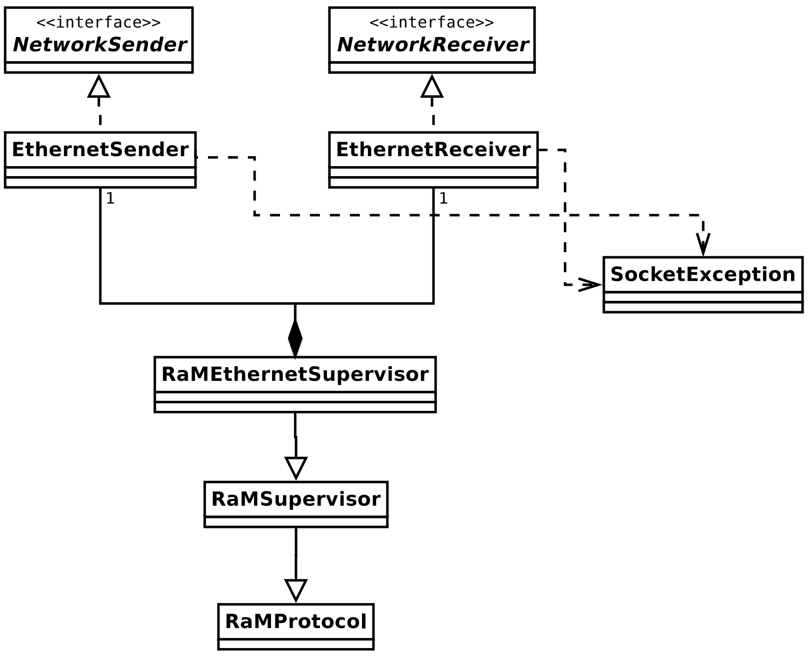


Figure 5.4: Class diagram of the Communication package.

CommunicationServer Package

The *CommunicationServer Package* contains two classes which regulate incoming and outgoing data. The RaMCommunicationServer class functions as a main class to configure, start and stop the sending and receiving classes. Both the *RaMProtocolReceiver* and *RaMProtocolSender* are implementation of the LUNA *Runnable* interface. This allows both to be executed as a separate thread. An overview of these three classes is given in Figure 5.4.

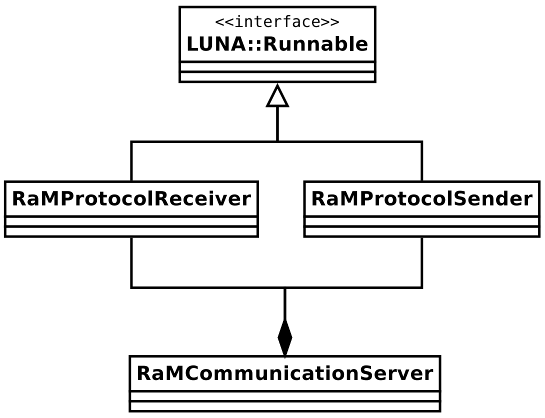


Figure 5.5: Class diagram of the Communication Server package.

The *RaMProtocolReceiver* class waits for data packets from the network. When it receives a data packet, it processes it and writes the values to the appropriate queues so the applications attached to the *Communication Component* can read the data. The flowchart for this process is show in Figure 5.6.

The *RaMProtocolSender* class creates a thread for sending data, which checks whether data is available from the applications that are connected to the *Communication Component*. For

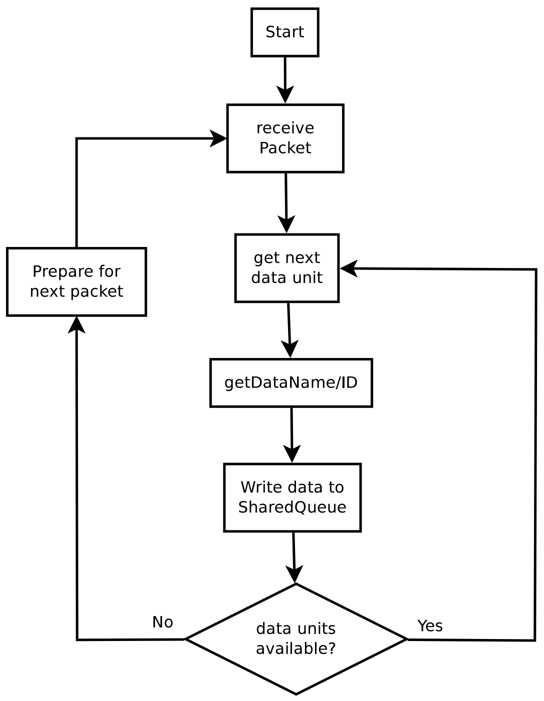


Figure 5.6: Flowchart of the receiving thread.

each data signal, a separate *Shared Memory Object* with a queue is created. Creating a single shared queue for each signal makes it easier for applications to connect to the *Communication Component* as each *Shared Memory Object* has a unique name. If data is available in a queue, it will send it to the appropriate destination, this is shown in the flowchart of Figure 5.7. The destination for each signal is defined in the *Communication Component*'s configuration.

A polling scheme is used in order to check if new data is available in the queues of the connected applications. This way different applications can be connected at different frequencies. This implementation is chosen as it fulfils Requirement 6. Downside of the polling approach is that an additional delay is introduced due to the polling frequency. The effect of this delay is illustrated in Figure 5.8. The worst-case delay of the signal is given by the polling period time of the *Communication Component*, plus the network delay and the polling period time of the receiving application.

Using the polling approach allows for asynchronous data communication next to the synchronous communication. This fulfils Requirement 9.

When the applications connected to the *Communication Component* need to exchange data on the same platform, the shared memory queues are used for the communication between those application. The *Communication Component* then facilitates in the setup of the queues (Requirement 10).

API Package

The API Package provides a class for applications to connect to the *Communication Component* and exchange data. The class is shown in Figure 5.9. The class constructor provides a way to connect to the *Communication Component*. Read and write function are included for data exchange. The class buffers the last received value from the *Communication Component*. If the queue is empty during a read call, the buffered value is returned. The current implementation of the class does not provide an indication that a buffered or new value is used.

The *SharedLockFreeQueue* class that is used for the data exchange, provides an indication that the queue is empty or not. This is used to determine if a buffered value should be used. The cur-

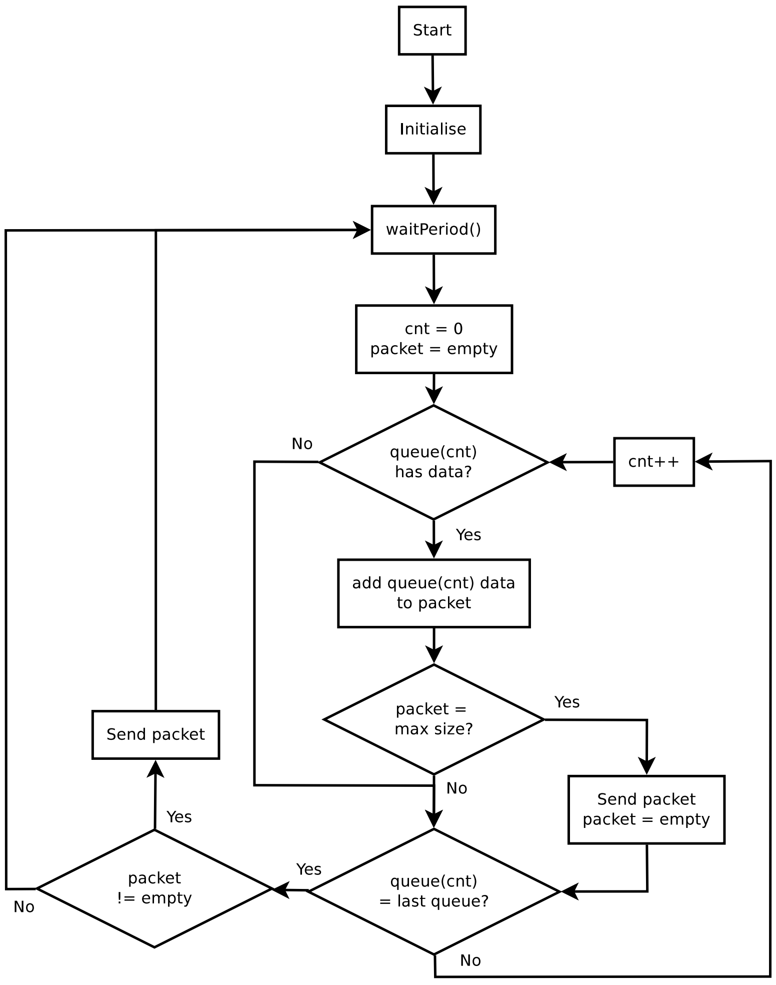


Figure 5.7: Flowchart of the sending thread.

rent class can be extended with a read function that also includes the indication of the *Shared-LockFreeQueue*, to indicate if a buffered value is used or not.

5.3.2 Server Configuration

The destination information for data provided by applications on a platform and the information regarding data that is received by the platform is stored in the *Communication Component*. Per platform the *Communication Component* stores a list of data signals that are send from the platform or received by the platform.

In order to provide this information to the *Communication Component* a configuration file format is defined. In the startup procedure of the *Communication Component* this configuration file is processed and the information is stored in the *Communication Component* on the platform.

XML is chosen as the encoding format for the configuration file. Using an XML Schema (The World Wide Web Consortium (W3C), 2005), a strict document structure can be specified which is easy to read by the application. For the processing of the XML configuration document, the RapidXML library is chosen. This is a fast and simple standalone open source library for parsing XML documents.

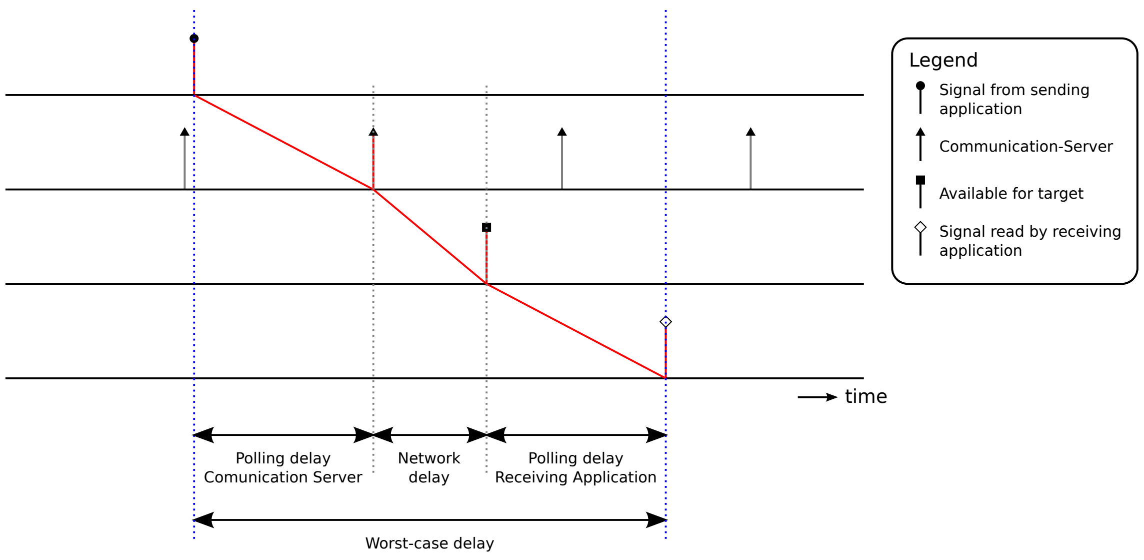

Legend
Signal from sending application
Communication-Server
Available for target
Signal read by receiving application
time
Polling delay Comunication Server
Network delay
Polling delay Receiving Application
Worst-case delay

Figure 5.8: Delay introduced due to polling implementation.

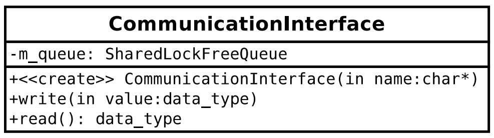

CommunicationInterface
-m_queue: SharedLockFreeQueue
+<<create>> CommunicationInterface(in name:char*)
+write(in value:data_type)
+read(): data_type

Figure 5.9: Class diagram of the API package.

5.4 Conclusion

Three parts are considered in the development of the *Communication Component*: The IPC, the *Communication Server* and *Network Communication*.

Several approaches have been considered for the implementation of IPC. However the downside of most of them, is that they rely on blocking read functions or interrupts. Therefore, a *Shared Memory Object* is chosen. In order to regulate access to the memory between processes a lock-free queue algorithm is implemented. This allows for applications to access the queue without blocking their own operation. In order to increase robustness, data from the *Communication Component* is buffered. The buffered data is provided if no new data is available when an application requests data.

Interfaces are defined for the *Communication Server* such that implementing new network protocols is easier, if these are required. Ethernet is used in the initial implementation of the *Communication Server*. LUNA is used for the implementation of platform-specific functionality, this allows for easier deployment across different platforms. XML is used to encode configuration information for the *Communication Component*. The RapidXML library is used for reading the configuration.

Several real-time network protocols are available and used in industry. However setting up these protocols requires specific hardware components or specific drivers. RAW Ethernet is chosen as the network protocol in the initial implementation of the *Communication Component*. Reason for this is that setups mostly consist out of one or two devices. Should more devices be connected in the network then a real-time Ethernet implementation should be chosen for the platforms connected to the network.

6 Results

In the previous chapter the design of the *Communication Component* is discussed. In this chapter the performance of the *Communication Component* is measured. Two experiments are described: A *Single-Signal* experiment to measure the latency of the signal in the *Communication Component* and a *Setup-Simulation* experiment, measuring latency performance based on the use case presented in Section 4.1.

6.1 The Experimental Setup

6.1.1 Hardware Components

The experimental setup consists of two Gumstix Overo Fires Computer on Module (COM) (Gumstix, 2013) with different baseboards. A schematic overview of the setup is given in Figure 6.1. The baseboards are the TOBI baseboard and the RaMstix baseboard. The TOBI baseboard is a commercial development board, the RaMstix baseboard is a board developed within the RaM group for usage as an embedded computer platform with the experimental setups. It features I/O similar to that of the TOBI baseboard, but is expanded with additional hardware components and I/O. Both Overo Fire COMs run a Linux OS with the Xenomai (Xenomai, 2013) real-time kernel, details are in Table 6.1. For the experiments, it is not relevant which baseboard is used for the computing platform.

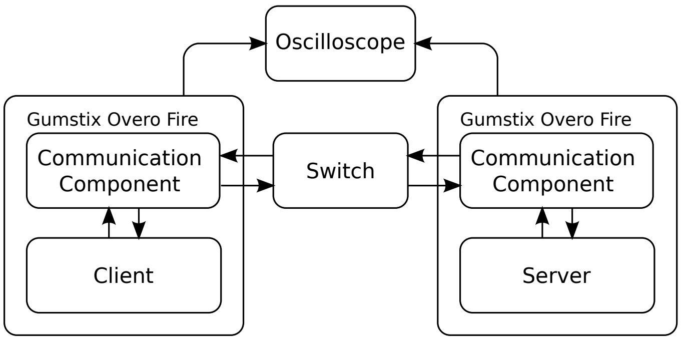


Figure 6.1: Schematic overview measurement setup.

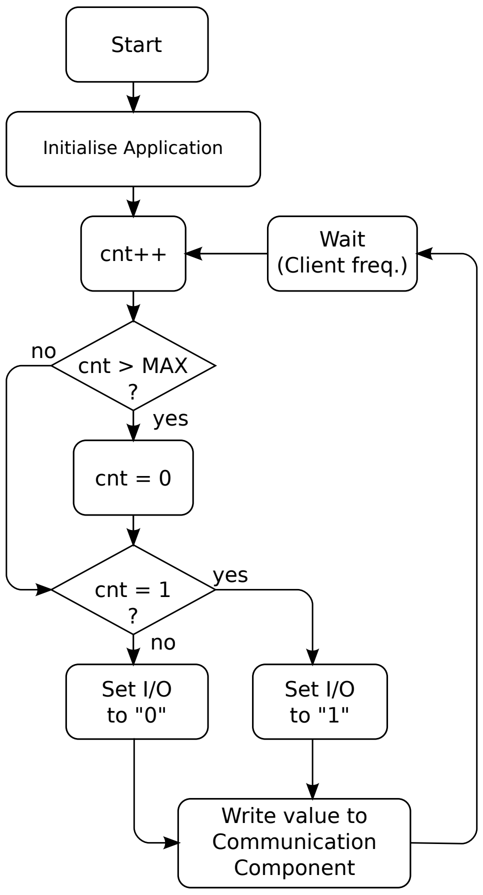
	Platform 1	Platform 2
Computer on Module	Overo Fire	Overo Fire
Baseboard	TOBI	RaMstix
Linux version	3.2.21	3.2.21
Xenomai version	2.6.2.1	2.6.2.1

Table 6.1: Measurement system specification.

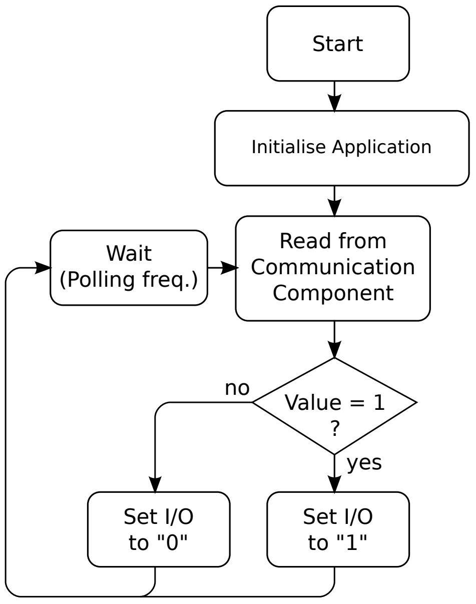
In order to connect the two boards, a switch is used rather than a direct connection. Due to the limited functionality of the Ethernet hardware on the baseboards, it is not possible to connect the two devices via a regular or cross cable. To measure the latency a general purpose I/O pin is used and connected to a digital oscilloscope. The I/O pin is toggled at the *Client*-device. The value of the I/O pin is then send from the *Client* to the *Server* application via the *Communication Component*. The *Server* sets the value of the I/O pin to the received value.

6.1.2 Software

Flowcharts of the *Client* and *Server* application are shown in Figure 6.2. The *Communication Component* is used to communicate between the *Client* and *Server* application.



(a) Client software flowchart



(b) Server software flowchart

Figure 6.2: Flow charts of the main measuring software.

The *Client* application toggles its I/O value and writes that value to the *Communication Component*. The frequency at which the value is written to the *Communication Component* is defined by the *Client Frequency*. To ease the readout on the digital oscilloscope, a delay mechanism is included in the application. Each period, a counter is increased, when the value of the counter is equal to one, the I/O pin is set to a "1", otherwise the output is "0". This is illustrated in Figure 6.3. This way, the delay of a previous signal does not interfere with the measurement. The value of the counter is send via the *Communication Component* to the *Server* application.

The *Server* application uses a polling scheme to read values from the *Communication Component*. When the received counter value is equal to one, the output of the I/O is set to "1".

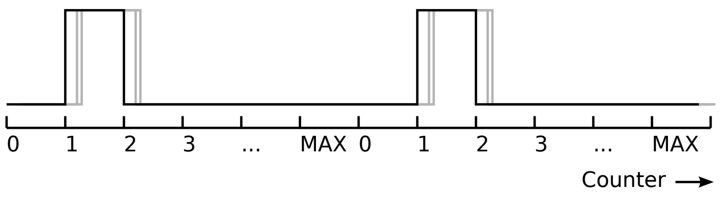


Figure 6.3: I/O signal delay graph.

6.1.3 Obtaining the Results

A digital oscilloscope is used to obtain the minimum and maximum latency. The persistency setting of the display of the scope is set to infinite, this allows to measure the jitter in the delay. The scope is set to trigger on the rising edge of the *Client* I/O signal, this way the delay of the *Server* signal can be read from the scope. The experiments are run for at least 10 seconds in order to get a good measurement of the minimum and maximum latencies. A result of a latency measurement is shown in Figure B.3.

6.2 Experiments and Expectations

6.2.1 Single-Signal Experiment

The *Single-Signal* experiment measures the latency when only one signal is send from the *Client* to the *Server*. This indicates the latency when processing is minimal.

As illustrated by Figure 5.8 in the previous chapter, using a polling scheme in the *Communication Component* adds a delay in the signal. The *Server* application also uses a polling scheme to obtain the data.

In the experiment, the expected worst-case latency is a combination of the polling frequencies of the *Communication Component* and the *Server* application, plus the network latency. In order to know the network latency an experiment is performed to measure this. The results are in Appendix B. The delay from the initial measurement of Appendix B is considered. This worst-case delay in the measurement is equal to 630 μs. The polling frequencies of the *Communication Component* and the *Server* application is 1000 Hz, the period time of both applications is 1000 μs. Adding this up leads to a worst-case latency of 2630 μs.

6.2.2 Setup-Simulation Experiment

In this experiment, a *Resource Intensive Algorithm Platform* and a *Control Algorithm Platform* are considered. In some situations it might be beneficial to obtain data from the real-time platform. For example, when mapping an environment with cameras, the position of the device may be required by the algorithm. This information needs to be send to the *Resource Intensive Algorithm Platform*. The results of the mapping algorithms then need to be send back to the *Control Algorithm Platform*. An overview of this is given in Figure 6.4.

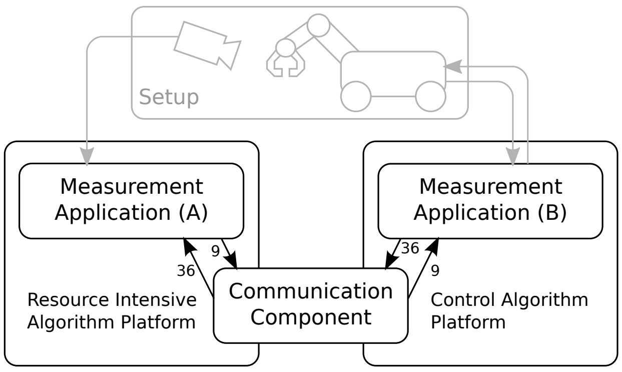

Setup
Measurement Application (A)
Measurement Application (B)
9
36
36
9
Communication Component
Resource Intensive Algorithm Platform
Control Algorithm Platform

Figure 6.4: Schematic overview of Setup-Simulation experiment.

A mobile robot with four wheels and a manipulator has been considered for the upper bound of the number of signals send between the *Resource-Intensive Algorithm Platform* and the *Control Algorithm Platform*. 36 signals are send from the *Control Algorithm Platform*, containing double float values. The *Resource-Intensive Algorithm Platform* sends 9 double float signals. Double float signals are chosen as these are 8-byte primitive values and result in larger data packets than using 4-byte data types.

Due to the increased amount of signals compared to the *Single-Signal* experiment, more processing is required by the *Communication Component.* Due to the use of embedded platforms in the test with limited resources, it is expected that the performance will get worse when transferring more signals.

6.3 Results

The measurements are performed with a digital oscilloscope with a dynamic sample frequency based on the display settings. The worst-case fault introduced by the scope in the measurements is 0.6‰. For the readout on the scope, the cursors of the scope are used. The worst-case fault introduced by the readout of the cursors is 1%.

In the *Setup-Simulation* test, one measurement is performed with a larger fault. In this measurement a fault of 4% is introduced. The measurement is indicated in the table (Table 6.4).

6.3.1 Single-Signal Experiment Results

The results of the *Single-Signal* experiment are shown in Table 6.2. The minimum and maximum latency are indicated, the last column contains the relative delay compared to the period time of the sending frequency. The relative latency is calculated according to Equation 6.1.

$$Relative\, latency = \frac{Max.\, latency}{Client\, freq.\, Period} \times 100\% \qquad (6.1)$$

Client freq.				
(μs)	(Hz)	Min. latency (μs)	Max. latency (μs)	Relative latency (%)
10,000	100	900	2000	20
5000	200	980	2100	42
3333	300	280	2140	64
2500	400	580	1800	72
2000	500	840	2120	106
1666	600	620	1880	113
1250	800	840	2020	162
1000	1000	400	1450	145

Table 6.2: Single-Signal measurement results. For the Client frequency, both the period time and the frequency are given.

6.3.2 Setup-Simulation Experiment Results

The *Setup-Simulation* experiment was performed twice. In the first experiment the latency from *Measurement Application* (A) to (B) is measured, this is shown in Table 6.3. In the second experiment the latency from *Measurement Application* (B) to (A) is measured, these results are shown in Table 6.4. In the measurements the data transfer frequency between the applications is altered.

6.4 Analysis and Conclusions

6.4.1 Single-Signal Experiment

The expected result for the *Single-Signal* experiment is that the worst-case latency would be at most 2630 μs. In the results of the experiment, the maximum latency that is measured is 2140 μs. This is within the range of the expected worst-case latency.

The *Communication Component* is designed for low frequency algorithms. The *Communication Component* performs well in the low frequency data transfer, as can be seen in the *Relative*

Transfer freq. (μs)	(Hz)	Min. latency (μs)	Max. latency (μs)	Relative latency (%)
10,000	100	600	2000	20
5000	200	900	3180	64
3333	300	960	2190	66
2500	400	1120	2600	104
2000	500	1260	2660	133
1666	600	900	2400	144
1250	800	920	3360	269

Table 6.3: Setup-Simulation results. Latency from Measurement Application (A) to (B). For the Transfer frequency, both the period time and the frequency are given.

Transfer freq. (μs)	(Hz)	Min. latency (μs)	Max. latency (μs)	Relative latency (%)
10,000	100	1400	1700	17
5000	200	550	2900	58
3333	300	1100	2250	68
2500	400	1000	2900	116
2000	500	920	2100	105
1666	600	1100	2280	137
1250	*800*	*950*	*5000*	*400*

Table 6.4: Setup-Simulation results. Latency from Measurement Application (B) to (A). For the Transfer frequency, both the period time and the frequency are given. The measurement of 800 Hz, was performed with a 4% fault.

latency. The *Communication Component* is not designed to be used in high frequency control, the *Relative latency* becomes worse at higher frequencies.

6.4.2 Setup-Simulation Experiment

The expectation for this experiment is that the additional processing required by the server would increase the latency. However for low data transfer frequencies, this is not the case. The *Communication Component* still performs well in the low frequency range. Also in the higher data transfer frequencies, the *Communication Component* still performs well, however it is not designed to be used with these frequencies, which is also indicated by the *Relative latencies*.

Should an application require more data to be send per period, an experiment measuring the maximum bandwidth should be considered.

7 Conclusions and Recommendations

7.1 Conclusions

The goals for this assignment are to design and implement a *Communication Component* to connect different platforms via a network. Provide a method for applications on those platforms to exchange data via the *Communication Component*. Finally, demonstrate the working of the *Communication Component* by means of a demonstrator.

7.1.1 Requirement Evaluation

By means of a platform analysis and a use case analysis, requirements for the *Communication Component* and requirements for an interface that enables applications to connect with the *Communication Component* are drafted. The requirements for the *Communication Component* are separated in two parts, the *Communication Interface* and the *Communication Server*. For both the requirements are evaluated.

A general requirement for the *Communication Component* is that it must be platform independent, as stated in Requirement 1. In order to achieve this, the *Communication Component* is developed using the LUNA framework. The LUNA framework provides an OS abstraction layer. Using the framework allows the *Communication Component* to be portable between platforms.

Communication Interface

- The *Communication Interface* provides an API in the form of a C++ class that allows to connect and to read and write data to and from the server. (Requirement 3)
- The Gumstix Overo Fire platform will be the deployment platform for all future experimental setups. However due to the fact that there is no Windows-based Compiler available for this platform, it is not yet possible to have it as a 20-sim 4C target. Therefore integration with 20-sim could not yet be tested. (Requirement 2)
- Integration for Simulink is neglected due to restricted time. (Requirement 5)

Communication Server

All requirements for the *Communication Server* have been met:

- Using a polling scheme, algorithms running at different loop frequencies can be served on the same platform. (Requirement 6)
- In order to make the application deterministic, all memory buffers are preallocated. For the encoding of the data packets, fixed widths for encoding of data values is used so that the encoding and decoding is deterministic. Also due to configuration, the amount of signals send over the network is known and therefore a maximum processing time can be determined. (Requirement 7)
- Using Ethernet, network communication is realised. (Requirement 8)
- The polling approach also allows for asynchronous data to be send by the server. (Requirement 9)
- For on target communication, the *Communication Component* provides the same IPC as it does between the *Communication Server* and the application. The *Communication Interface* connects to the correct queue for the data exchange. (Requirement 10)

Non-Functional Requirements

Tasks performed by the server have been kept to a minimum, on the sending side it polls for data and sends it when it is available. On the receiving side, it only executes when a data packet arrives at the server (Requirement 11). A configuration file format is defined for easy deployment. This makes it easy to define in- and outgoing data signals in the *Communication Component* (Requirement 12).

7.1.2 Demonstrator Results

To test the *Communication Component*, two experiments are conducted as a demonstrator. The results of these experiments show that the *Communication Component* has a stable latency for the loop frequencies it is designed for.

7.2 Recommendations

Setting up a tool-chain with computing platform for an experimental setup takes a lot of time and knowledge. In order to deal with this, within the RaM group a generic computing platform is created in the form the RaMstix. However the RaMstix platform does not yet have support for the current deployment tools, such as 20-sim 4C. 20-sim is used a lot within the group for the design of control law. Therefore a 20-sim 4C target should be made for the RaMstix computing platform to ease the deployment step in the way of working and encourage the usage of 20-sim as a design tool for control within the group.

The Simulink/Matlab tool provides toolboxes for prototyping developing image processing algorithms. However integration with the applications is not yet possible with *Communication Component*. A interface should be made for Simulink/Matlab in order to integrate the functionality of these tools with the *Communication Component*. This will benefit the development process by making integration of image processing algorithms easier.

The OROCOS framework provides building blocks to create complex supervisory and sequence controllers. It is already used in some of the setups that are not reviewed within this thesis. Integrating these building blocks to the real-time loop control platforms would be beneficial for development, however no integration exists yet. Integration with the OROCOS framework should be provided, this could be by including a component in OROCOS that can connect to the *Communication Component* or by extending the *Communication Component* to include the network protocol used by OROCOS.

A Domain Analysis

A.1 Introduction

The domain analysis will focus on two main aspects: the software tools used in the experiments and the way of working in the experiments. Only a few setups are considered in this research as it is not feasible to analyse all the setups. The setups are a representation of most of the fields in which the groups do research and are currently used in active research projects. Educational setups are left out of the analysis as these are not the focus of this thesis. The setups included in the domain analysis are:

- Bipedal Walker
- Variable Stiffness Actuator (VSA) UT II
- Parallel bars
- Microrobotic setup
- Flexible needle setup

First the software tools and hardware components common in the experiments are introduced, these are followed by the analysis of the different experimental setups. The analysis will conclude with observations and conclusions.

A.2 Software Tools and Hardware

A.2.1 Software Tools

There are a number of tools which are used in several projects. These tools are for the design and deployment of the control law designs. These tools are Matlab/Simulink from Mathworks (Mathworks, 2013), 20-sim and 20-sim 4C from Controllab Products (Controllab, 2013)

Simulink/Matlab for Controller Design

The Matlab/Simulink tool is used for controller design. Matlab is a tool for numerical computation and programming. Simulink extends Matlab with a block diagram environment which can be used to create models of dynamic systems. In Matlab there are toolboxes available specifically for design and analysis of control systems.

Simulink for Deployment

Matlab/Simulink can also be used for deployment. The tools can be extended with code generation toolboxes. These toolboxes can be used to generate generic or target specific code from models or functions in the tool. In order to create target specific code, blocks of the hardware components have to be included within the models.

In addition to the code generation, the tools also provide rapid prototyping facilities. There are two approaches to rapid prototyping provided by this tool. The first is a Personal Computer (PC) in the loop solution, and the second a hardware in the loop solution.

In the PC in the loop solution, Simulink uses the Real-Time Windows Target feature. This feature launches a real-time engine which runs in Windows kernel mode. In this engine the device drivers are loaded. In normal running mode, the model is run in Simulink and the outputs of the model are connected to the device drivers in the real-time engine. In external mode, the code generation toolboxes are used to generate an application which is also loaded into the real-time engine. Simulink then only acts as a monitoring tool.

When using the hardware in the loop solution, a dedicated target is used to control the setup. The target runs the xPC real-time kernel which is developed by Mathworks. The code generation toolboxes are used to generate the control application for the target based on the models. Simulink connects with the target via an Ethernet connection and acts as a monitoring platform.

Both solutions also offer the capability to perform online modification of parameters. This means that for parameter changes, the entire application does not have to be regenerated and recompiled for the target.

20-sim for Controller Design

20-sim is a modelling and simulation tool. It is specially designed for the analysis and design of dynamic systems. The tool comes with various toolboxes which aid in the the analysis and the design of controllers.

20-sim 4C for Deployment

20-sim offers a deployment method by generating code from the submodels in the system. In addition it can generate code that can be imported into the 20-sim 4C tool. The 20-sim 4C tool is a rapid prototyping tool. Apart from importing 20-sim code, 20-sim 4C can also import code from other tools if the code is compatible with the 20-sim 4C tool.

20-sim 4C offers an interface which allows to user to connect the ports of the submodel to the various hardware components of the target. The available components of the target are described in a target configuration file. The usage of configuration files allows 20-sim 4C to work with generic targets.

For communication with the target Ethernet is used in combination with a daemon application that runs on the target. Currently only the following operating systems are supported for the target:

- Windows, Linux (non real-time)
- RTAI Linux (real-time)
- Xenomai Linux (real-time)

20-sim 4C offers features to monitor the experiment and perform online modification of parameters.

Conclusions

Both Mathworks and Controllab Products offer a rapid prototyping solution. Simulink offers both a PC in the loop and a hardware in the loop solution and Controllab a hardware in the loop solution.

The tool-chain of 20-sim is more generic. The models don't contain implementation specific parts, these are added in the deployment step. With Simulink, the hardware components have to be added to the model before generating the code for the application for a specific target. This means that in the 20-sim 4C tool there is a better distinction between the controller design and the implementation in comparison to Simulink.

A.2.2 Hardware Components

Motor Control

The "Solo Whistle" is a commercial controller created by ELMO Motion Control (Elmo Motion Control, 2013). It is an external board which has the capability to control a motor via current, velocity or position control. It contains input on which encoders can be connected and also comes with power electronics to power the motor. To control the motor setpoints can be send to the controller via a serial connection or a CAN bus.

Another controller solution is the FPGA in combination with an H-Bridge amplifier. The FPGA contains a counter and a pulse width modulator. Depending on the choice of the developer the controller can also be included in the FPGA or an external processor.

The “Solo Whistle” controller is used in the research projects, while the FPGA solution is used in the educational setups. The reasons why the “Solo Whistle” is used in the research project is because:

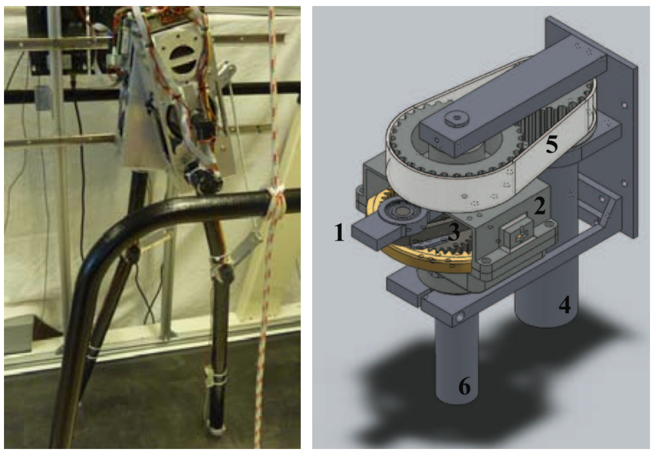
- The “Solo Whistle” is ready to use.
  There is no need to develop the loop controller for the motor or to determine how to implement it. Everything is part of the board.
- The “Solo Whistle” takes care of real-time constraints.
  There is no need to verify that the controller functions accordingly, the package is already tested by the manufacturer.
- The loop control law is not part of the research.
  For the educational setups, the loop control logic is sometimes part of the project, to make choices for controller placement in the design space exploration. For the research, the low level controllers are not part of the research. The “Solo Whistle” takes care of the low level loops, so the developer does not have to.

In one of the research projects, the “Solo Whistle’s” current control feature is used to control the magnetic field of magnetic coils rather than control a motor.

A.3 Research Setups

A.3.1 Bipedal Walker and Variable Stiffness Actuator

The Bipedal walker setup and the VSA UT II setup can be considered as similar kind of setups based on their software structure. The differences between the setups is their mechanical design and the goal of the setups.



(a) Bipedal Walker. **(b)** VSA UTII.

Figure A.1: The Bipedal Walker setup (a) (Geus, 2012) and the VSA UT II (b) (Groothuis, 2011).

Bipedal Walker

Human walking is very energy efficient and robust. Humans are capable of changing the stiffness of their joints during walking. In order to improve robotic walking, patterns of human walking are mimicked in the design and control of robotic walkers (Geus, 2012). The bipedal walker (Fig. A.1a) was created to do research on the effect of having variable stiffness in the joints of the robotic walker.

The bipedal walker setup consists of a gait which is positioned on a treadmill, this way limited space is required for the experiments. In order to limit the freedom of motion for the setup to the sagittal plane, the bipedal walker is attached to a guidance.

Experimenting on the Bipedal Walker

The control law for the experiment was designed using the 20-sim tool. For the deployment an NXP MBED (NXP, 2013) processor board was chosen to run the control law. The control law was implemented by means of a C-application and was executed on the NXP MBED board. A PC was used to monitor the experiment and to send commands to control the experiment.

To perform an experiment on the bipedal walker, there are a number of steps which have to be done. These steps are:

1. Hoist the robot so it is free from the floor.
2. Initialise all components (Motor controllers/Connections/etc.).
3. Perform the homing procedure, so that all encoder counters are zero.
4. Start the experiment, the robot will go into its start position.
5. Lower the robot onto the treadmill.
6. The robot will start walking after a short period once it is placed on the treadmill.

Due to how the software was organised, the whole process described above had to be repeated for each single experiment. This is very inconvenient if the developer wants to conduct multiple experiments. In addition for every new controller, a new c-application has to be compiled in order to implement the controller.

In order to deal with some of these problems, the software was restructured such that the controller was no longer executed on the NXP MBED board. The board was now used as an interface between the PC and the hardware. The control law was moved to the PC and run in Simulink using the Real-Time Windows Target feature. In this new structure the hardware initialisation and the homing procedure only had to be executed once.

The NXP MBED solution was chosen at the time as it seemed like an adequate board. It is a cheap processor board with a relatively powerful ARM microprocessor. Also all the necessary I/O was available on the board. However the board is not capable enough to run for example a real-time operating system, this limits the boards capabilities in the use with tool-chains.

Variable Stiffness Actuator UTII

Humans have the capability to vary the stiffness of their joints, this means they can adapt to their surroundings. In robotics, actuators with stiff joints are mostly used, this is ideal for position control but not for interaction with the environment (Groothuis, 2011). It is therefore desired to have actuators with variable stiffness when interacting with the environment. The VSA UT II was developed to research a new rotational VSA design which actuates a lever. In continued research, additional applications next to actuation of the VSA are explored.

Experimenting on the VSA UTII

The control law for the VSA UT II is designed using the 20-sim tool. For the deployment an Arduino Mega board Arduino (2013) is used on the setup. The processing power is limited when executing more complex models, therefore the same software restructuring was done as with the bipedal walker. The control law is exported and imported into Simulink model. Simulink with the Real-Time Windows target is used to execute the control law for the experiment. The Arduino is used as an interface between the hardware and the PC with Simulink.

The experiment itself does not take as many steps as bipedal walker as the setup is stationary. An absolute encoder is used for the position feedback. The setup is put into a default position from which the experiment can be started.

Setup Structure

The structure of both setups is similar and is shown in figure A.2. A PC is connected to a microcontroller board via a serial connection. The microcontroller board contains general I/O which is connected to the various sensors of the setup. For the control of the motors the "Solo Whistle" is used. These controllers are connected to the microcontroller via a serial rs232 interface or CAN bus.

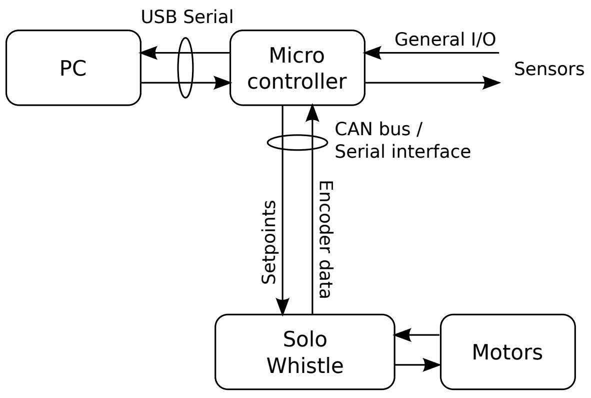


Figure A.2: Bipedal walker and VSA UT II setup architecture.

On both setups the control law is implemented in a Simulink model. The model is executed as a Real-Time Windows target (Ketelaar, 2012). The microcontroller is used as an interface between the hardware and sensors, and the PC with the Simulink model.

Conclusions and Recommendations

On both setups a microcontoller is used as an interface between the hardware and the PC. Initially the microcontrollers were used to execute the control application. This is very inconvenient because the developer has to write its own application for the microprocessor board. For every different experiment a new application has to be created.

In addition the computation power of the microcontroller boards is limited. This means that the sampling time of the controllers is limited by the computation power of the boards. This also limits the execution of more complex control algorithms on the boards.

The recommended solution would have been to add an (embedded) computer with a real-time Operating System (OS). A real-time OS already has software implementations for use in real-time applications which makes development time of applications shorter. Also some of the tool-chains can be integrated with an embedded computer, making the experimentation process easier.

Another issue in the way of working is how the tooling is used. 20-sim is used to design and test the controller using models. The controller is then exported and imported into Simulink. For the user this means he has to be familiar with two different tools. Simulink is used because it can run the controller model in real-time with the microprocessor board using the Real-Time Windows Target feature.

In addition with the previous recommendation, using an (embedded) computer with a real-time OS would have given the possibility to use the 20-sim 4C tool. This would have removed the need to use two different tools.

A.3.2 Parallel Bars

The parallel bars setup is used to research the effect of networks in distributed control systems. Effects are for example varying time delays or message loss. Initially the parallel bars setup was build to verify a proposed controller design by Franken (2011).

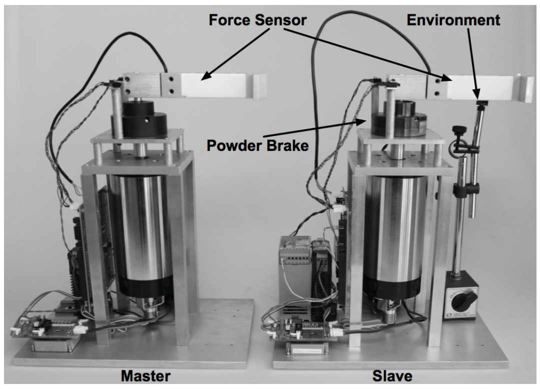


Figure A.3: Parallel bar setup. (Franken, 2011)

The setup (Fig. A.3) consists of two similar devices. Each device contains a motor which actuates a lever. Attached to the lever is a force sensor for feedback.

Experimenting on the Parallel Bar

The controller is designed in 20-sim and simulated using a model of the plant. The control law is deployed on an embedded system running a real-time Linux operating system. In order to load the control model on the target 20-sim 4C is used. In 20-sim 4C there are a number of steps which are needed in order to perform an experiment:

1. Select the component to use on the target, in this case the control block.
2. Connect the ports of the control law block to the corresponding hardware components, like the encoder sensors or actuators.
3. Generate and compile the code for the target.
4. Select which parameters of the model to monitor.
5. Perform the experiment.

Setup Design

Figure A.4 shows the architecture of the parallel bar setup. The architecture is similar to that of the Bipedal Walker and VSA UT II setups. In this setup however the microcontroller is an embedded computer system which runs a real-time OS. The embedded system is connected to a PC via an ethernet connection. The motors are controlled by the“Solo Whistle” which are connected to the embedded system via CAN bus.

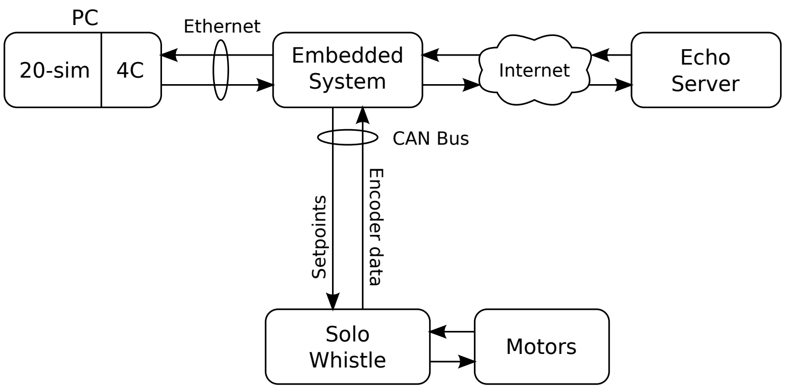


Figure A.4: Parallel bar setup architecture.

The network of the setup is contained within the control model. The network behaviour is simulated, or a physical network is used with an echo server. The echo server returns the messages it receives to the original server. This allows the control system to send its messages to itself via a physical connection.

Conclusions

In the parallel bar setup the tool-chain provided by Controllab is used. The tool-chain consists of 20-sim for the design and modelling of the system and 20-sim 4C for the deployment. This way of working is a good practice. The developer is not required to do any programming or other manual work. The controller design is exchanged between the two tools and no manual actions are required in this step. The final application is created using code generation.

The network of which the effects are researched is not part of the 20-sim or 20-sim 4C tool. This part is integrated using the feature to include custom code within the real-time application.

A.3.3 Microrobotic Setup

In the medical area, research is done into the field of minimally invasive surgery. One of these projects is regarding the use of microrobotics. These microrobots can for example be used to deliver drugs at specified locations in the human body. The microrobots can be controlled using magnets and are observed by means of a camera. Goal of the research project is to demonstrate that microrobots and particles can be controlled in a 3D space with the use of magnetic fields.

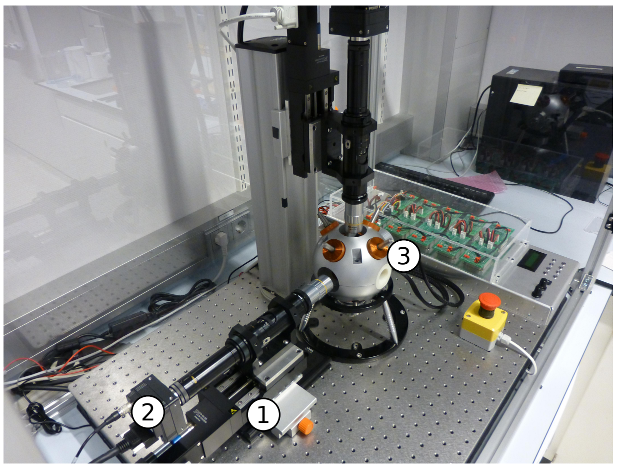


Figure A.5: Microrobotic setup. (Khalil et al., 2013)

The setup consists of a test environment in which the microrobot or particle is placed. Around this test environment magnets are placed for creation of the magnetic fields (Fig A.5 (3)). For feedback on the position of the microrobot cameras are placed (Fig A.5 (2)). Autofocus of the camera on the particle is achieved by moving the cameras towards or away from the environment by means of a motor (Fig A.5 (1)).

Experimenting on the Microrobotic Setup

The controller for the experiments is designed using the Matlab/Simulink tool. For the deployment of the control application the xPC target platform from Mathworks is used. The Matlab/Simulink tool-chain was chosen instead of the 20-sim/20-sim 4C tool-chain because Matlab contains toolboxes for image processing, that were required for processing the camera images. From the Simulink model, code is generated with the code generation toolboxes and compiled for the xPC target.

The setup does not require a specific homing procedure. The position of the cameras is controlled by the autofocus algorithm and is not dependant on an initial location of the camera.

Setup Design

The setup consists of two PCs: One PC is used to run the Simulink application, the second PC is the target PC. This target PC was specifically configured to be used as a real-time target. On the target runs the xPC real-time kernel by Mathworks (Mathworks, 2013). The two computers are connected using an ethernet connection. The “Solo Whistle” motor controllers and the camera are connected to the target PC, see figure A.6. The ELMO controllers are also used to control the magnets in the setup, using the current control feature.

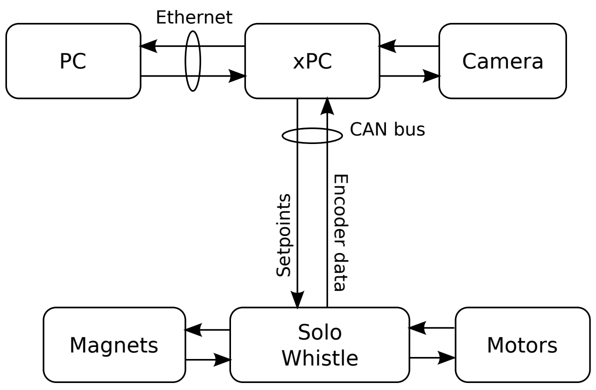


Figure A.6: Setup architecture of the micro robots setup.

Conclusions

The tool-chain provided by Mathworks with Simulink and xPC target is used in the microrobotic setup. In the tool-chain it is not necessary to do any manual programming work. The application for the target is created using code generation. The tool provides monitoring capabilities and online parameter modification.

A drawback of this tool-chain is that the hardware support limited to a restricted number of vendors. Also deployment has to be built into the models, making models hardware specific. However experimenting requires less effort. The tools provide capabilities to generate and compile the code for the deployment target based on the model. The tools also provide monitoring and control capabilities.

A.3.4 Flexible Needle Setup

Another medical research project is the flexible needle. This is also a research project in the field of minimally invasive surgery. For certain types of surgery, needle injection is used to perform the surgery. A flexible needle allows the surgeon to control the path of the needle. For position feedback, ultrasound imaging is used. A path planning algorithm is provided for this research project in the form of a C++ class from another university. The goal for the research is to develop and improve the steering of a flexible needle into the tissue.

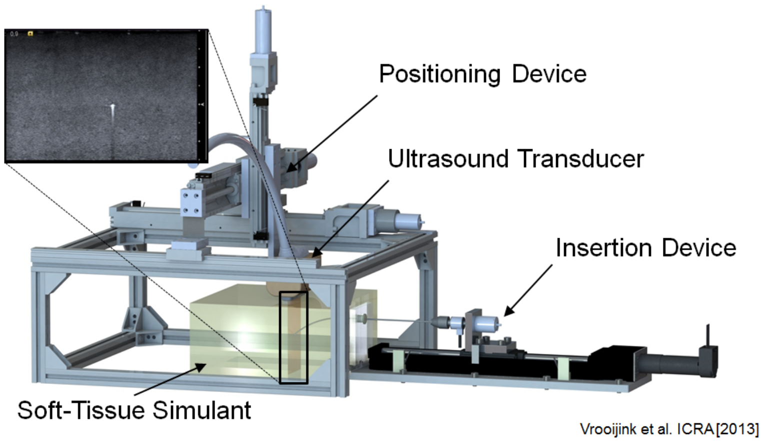


Figure A.7: Flexible needle setup. (Vrooijink et al., 2013)

The setup consists of a few key components (Fig A.7). A positioning device for positioning the ultrasound equipment on the tissue simulator, ultrasound transducer for the visual feedback and an insertion device. The positioning device and insertion device are controlled using motors.

Experimenting on the Flexible Needle Setup

The control law is designed in either Simulink, 20-sim or another tool based on the preference of the developer. After the design the control law is included into a framework. The control application is deployed on a PC.

The framework was developed internally for the setup. The framework is written in C++ and contains classes for the device drivers of the system and some procedures, for example a homing procedure. The developer needs to manually compose the application with all the components and manually include the control algorithms. For each experiment a new application has to be created. After the developer has written the application, it is compiled and executed on the PC.

Within the application there are two main loops. The first loop is a soft real-time control loop in which the image feedback is used to control the motors. The images are provided by the ultrasound equipment and are not guaranteed hard real-time, therefore this loop is not hard but soft real-time. The second loop is for the path planner. Based on the current position and obstacle placement, this algorithm calculates the path for the needle in the tissue simulator.

Setup Design

Figure A.8 shows the structure of the flexible needle setup. The PC is connected to the motor controllers by means of a CAN bus. The ultrasound equipment is also attached to the PC. In this setup there are no additional embedded computers or microcontrollers used.

The PC runs a Windows operating system. Windows is a best effort operating system and not real-time. This means that there are not guarantees for the real-time constraints. This some-

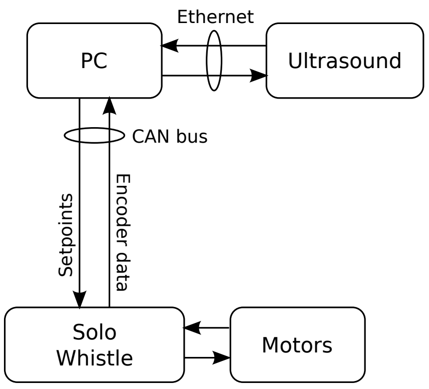


Figure A.8: Flexible needle setup architecture.

times lead to timing issues when additional software was activated on the PC. In order to cope with this, all unnecessary software is turned off during experiments.

Conclusions and Recommendation

Windows is chosen as the deployment platform for this setup. The downside of Windows is that it is a best effort OS, this means that there are no guarantees for real-time constraints. A better solution would be to divide the various software components among a real-time platform and Windows platform. This way the different real-time layers can be separated.

In the deployment step, the control application is manually composed with use of the developed framework. The framework allows for reusage of most of the code in the application. However due to the fact that composition of the code is a manual process, mistakes can be made. Using code generation from the development tools and applying those where possible could reduce the process of manual code creation. This would reduce the chance for errors and could also speedup the experimentation process.

A.3.5 Software Structure

Figure A.9 shows the generic structure of an Embedded Control System (ECS) as developed within the Robotics and Mechatronics (RaM) group (Broenink et al., 2010). In this section the tool-chains used in the setups are compared on how they map on this structure. In the comparison, two different implementations can be identified. The setups which use Simulink with the Real-Time Windows target have a different implementation than the setups which use 20-sim 4C or the xPC target.

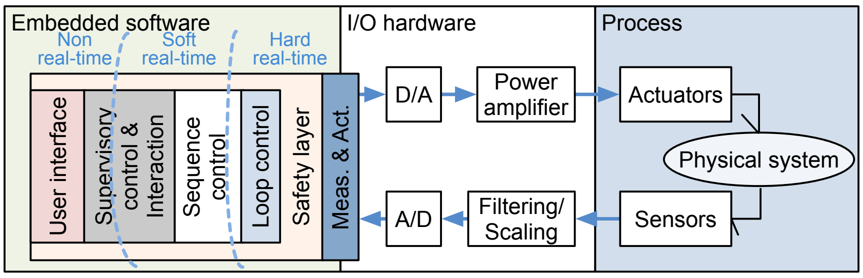


Figure A.9: Generic software structure of an embedded control system. (Broenink et al., 2010)

Simulink with Real-Time Windows Target

Figure A.10 shows the implementation of the generic software structure in the Real-Time Windows Target. The hard real-time loop control is executed on the "Solo Whistle". Between the loop control and the higher order control logic is an additional interface layer. It allows the Simulink model to communicate with the hardware components connected to the microcontroller board. The higher order control loops are run on the PC on a best effort Windows OS.

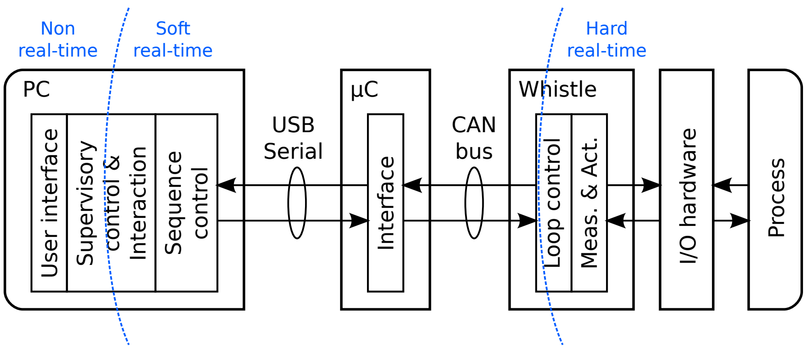


Figure A.10: Software structure with Simulink and Real-Time Windows target.

For this a real-time engine, Real-Time Windows Target, is run in Windows kernel mode. Due to the OS being best effort, the control loops are not guaranteed real-time.

Simulink with xPC Target and 20-sim 4C

Figure A.11 shows the generic stucture implemented in the parallel bar and microrobotics setup. The hard real-time loop control is executed on the “Solo Whistle" like the Real-Time Windows implementation. For the higher order control loop a dedicated target is used, that runs a real-time OS/kernel.

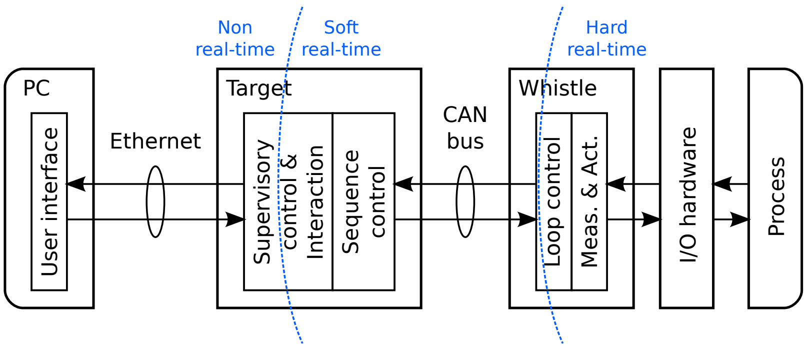


Figure A.11: Software structure for setups with a target PC or embedded system.

The PC is used to monitor the application and is not real-time. The separation between the different real-time and non real-time layers is clearer in this tool-chain. Real-time loops run on targets which are designed to run those loops and the non real-time Windows OS is used only for monitoring and online parameter modification.

Conclusions

The tool-chain with the Real-Time Windows target offers a best effort solution. Mostly deadlines are met if the computer is fast enough. The tool-chain provided by 20-sim/20-sim 4C and Simulink/xPC target is preferred, since they have a better seperation of the hard/soft real-time layers and the non real-time layers. Also the real-time loops are run on dedicated machines with real-time OS.

A.4 Conclusion

Both Mathworks and Controllab products offer tool-chains which simplify the experimenting process by removing the need to write custom code in order to implement control law algorithms. Two experimental setups within the RaM lab already successfully use these tool-chains in order to conduct experiments. Some other setups use the Real-Time Windows Target tool-chain in combination with a microprocessor board which acts as an interface. These latter setups could benefit from applying a full tool-chain in which a dedicated real-time deployment target is included to run the control law, rather than the limited microprocessor board which is used now.

Another observation is the fact that the robotic setups are not just a dynamic system with sensor feedback and a control loop. Specially in the medical setups image processing algorithms with cameras are used within the feedback loops. Image processing is a computational intensive task which sets now requirements for the hardware of the setups in terms of processing power.

B Ethernet Latency Measurement with Gumstix Overo Fire

B.1 Introduction

For control loops it is important to know the latency of the control signals and the feedback signals in the system as this affects the overall performance of the system. Ethernet networks are now implemented more as communication interface in control systems. There are several real-time protocols available which can be used in control systems, however these often require specific hardware or drivers.

This measurement aims to measure the latency of standard ethernet on an embedded target. For the transmission of data packets the UDP/IP protocol is used as the sending and receiving protocol. The measurement should give more insight into the performance of Ethernet under certain conditions and specifically under circumstances for the experimental setups in the RaM laboratory. Most often these use private networks or direct connections, so the effect of connecting the system to the Internet is neglected.

B.2 The Measurement Setup

B.2.1 The Hardware Components

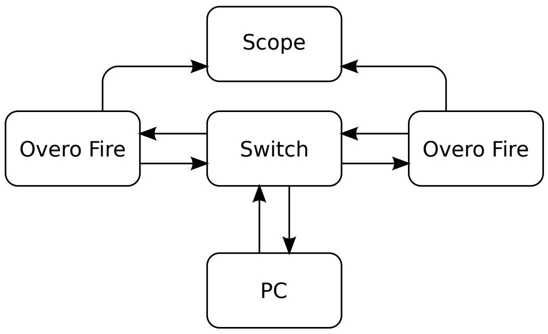


Figure B.1: Ethernet setup overview.

Figure B.1 shows a schematic overview of the measurement setup. The setup consists of two embedded targets, in this case a Gumstix Overo Fire Computer on Module (Gumstix, 2013), details of the embedded target are in table B.1. In the setup, one Overo Fire will act as the client, the other will act as the server in the measurements.

Computer on Module	Gumstix Overo Fire
Baseboard	Tobi
OS	Linux, kernel version 3.2.21

Table B.1: Embedded computer target details.

In order to measure the latency and not be dependant on the synchronisation of a network clock, a scope is used to measure I/O signals on the embedded targets. When a packet is send from the client an I/O pin is toggled, as soon as the packet is received on the server, an I/O pin is set to the same value as indicated by the client packet. This results in a delayed signal on the server. The scope is used to measure the delay between the two I/O signals.

Additionally a Personal Computer (PC) is included in the setup to act as an additional source of network disturbance. In the test, two sample frequencies are tested for the I/O pin. 100 Hz and 1000 Hz.

B.2.2 Communication Protocol

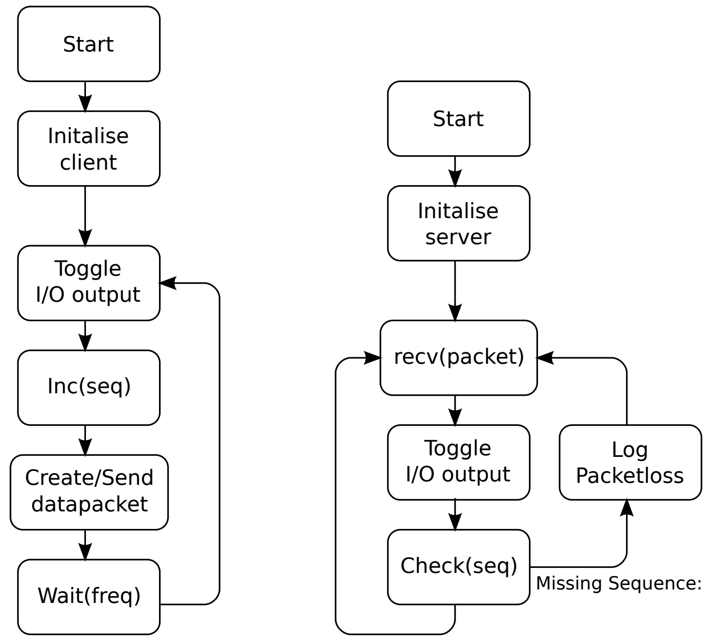
For the communication between the client and the server UDP/IP is used. This protocol sends datapackets called datagram packets. The datagram packets are created so that they are a total of 128 bytes in the first set of measurements. This size is chosen as a number of sensor values could fit into these packets (Dolejs et al., 2004). The headers of Ethernet and UDP/IP ar included in these 128 bytes, however the Preamble and “Start of frame delimiter” are excluded in this set of bytes. This results in 42 bytes for the headrs and a payload of 86 bytes for the actual data.

In the datagram packets from the client a sequence number and the value for the I/O pin are included. The sequence number is used to check if packet loss occurs on the connection. The I/O pin value is used for setting value of the I/O on the server. Should a packet be lost, than this does not result in a 180 degree phase shift in the server signal.

The application producing additional load on the network also sends 128 bytes per packet, similar to those of the client application. The load packets are send to a different port, than the server uses to listen. This way the server does not need to filter the load packages from the client packages.

B.2.3 The Measurement Software

For the communication via Ethernet, the UDP/IP protocol is used. Data packets send by the UDP/IP are called datagram packets. Due to the nature of this protocol, datagram packets can be lost. In order to detect this, a sequence number is added to the packet, in order to verify at the server side if a packet is lost.



(a) Client software flowchart

(b) Server software flowchart

Figure B.2: Flow charts of the main measuring software.

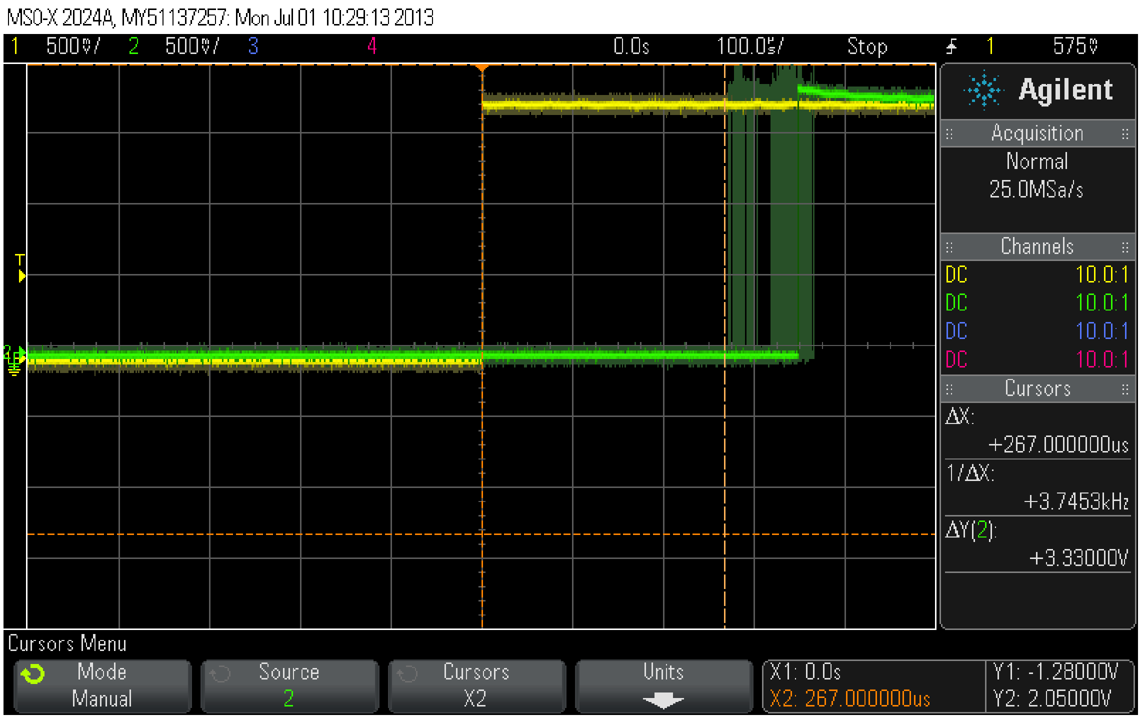
Figure B.2a shows the flowchart of the client application. The I/O toggle action is performed before creating and sending the datagram packet, this way the processing time of creating and sending the datagram packet is also included in the latency. The client keeps a counter *seq* in order to provide a sequence number for the datagram packets. The wait function makes sure that the packets are send at the predefined sample frequency of the system.

The server application flowchart is shown in figure B.2b. When a packet is received, the I/O output will be set to the value indicated in the datagram packet. After this, the sequence number will be checked. If the sequence number is not the expected value but a higher value, the amount of missed datagram packets is notified in the console.

In addition to the client and server application, there is also a load application. This application sends a burst of # packets in a set frequency in order to create additional load on the network.

B.2.4 Obtaining the Results

A scope is used to obtain the minimum and maximum latency in each test. In order to retrieve this data, the scope's display settings are set to infinite persistency. When a measurement is started, it is run for several seconds. Figure B.3 shows one of the measurements, the cursors are used to obtain the exact value for the minimum and maximum latencies.



MSO-X 2024A, MY51137257: Mon Jul 01 10:29:13 2013

1 500mV/ 2 500mV/ 3 4 0.0s 100.0us/ Stop 1 575mV

Acquisition: Normal 25.0MSa/s

Channels: DC 10.0:1, DC 10.0:1, DC 10.0:1, DC 10.0:1

Cursors: ΔX: +267.000000us; 1/ΔX: +3.7453kHz; ΔY(2): +3.33000V

Cursors Menu: Mode Manual; Source 2; Cursors X2; Units

X1: 0.0s; X2: 267.000000us; Y1: -1.28000V; Y2: 2.05000V

Figure B.3: Example of a result produced by the scope. The yellow signal is the signal produced by the client application, the green signal is the signal produced by the server. The scope is triggered on the rising edge of the client signal.

B.3 Measurements and Expectations

In this section, each of the experiments is explained, along with the expected result of the measurement. In Dolejs et al. (2004) simulations have been performed with an Ethernet network, along with a measurement experiment in which the Real-Time Publisher Subscriber (RTPS) protocol is tested. The values within the expectations are based on the findings of this paper.

The simulation states that network traffic below the 30,000 packets/s will result in low latencies in the order of 100 μs. This however does not include the processing time of the devices. When the network load is increased to 40,000 packets/s or above, higher latencies up to a few milliseconds are measured in the simulations.

In the real-world test in which the RTPS protocol is measured, the minimum latency is 0.7 ms and the maximum latency is 1.34 ms with a mean of 0.84 ms. Because a protocol on top of

UDP/IP is tested, the expectation is that the latency in the measurements will be less than the results of Dolejs et al. (2004).

B.3.1 Initial Measurement

In the initial measurement the client will send one packet with the I/O information to the server. This test is used to measure the latency, when no additional load is put on the network.

Taking into account that there needs to be some processing time, the expectation is that the latency will be higher than the 100 μs as presented in the simulation of Dolejs et al. (2004), however a protocol is not implemented so the latency should be less than the 0.7 ms.

B.3.2 High Traffic from the Client

In this measurement, the effect of high traffic from the client is measured. In order to make sure that the server is not affected by the high amount of traffic, the load packets will be send to the PC.

The expectation for this measurement is similar than that of the previous measurement. Because only one node in the network is sending data, collisions should not have an effect on the network latency. However the latency is expected to be higher than that of the initial test, sending more packets from the client means the client has to perform more tasks, this could slow down the performance, as all the packages have to be created.

B.3.3 High Traffic to the Server from the Client

This measurement will be similar to the previous measurement, however the load packets are now also send to the server.

Because only one node is sending, the expectation is that there will be no significant delay as collisions will not occur. The delay will be as with the previous measurement due to the amount of processing performed by the client, and in this case the processing of the incoming packets on the server.

B.3.4 High Traffic to the Server from a Second Source

This measurement will test the effect of a second source sending high amounts of data to the server and how this effects performance. In this test the client will only send the I/O control packet to the server, while the PC will send load packets to the server.

Because collisions can occur, the expectation is that on high traffic load, the latency of the network will increase in comparison to the other test. Based on the simulations presented in Dolejs et al. (2004), when the load will be below 30,000 packets/s, the latency will be about the same as in the initial test, but at higher load the latency will increase tremendously.

B.3.5 High Traffic to the Client from the Network

In this measurement high amounts of traffic will be send to the sending client node in the network from the PC node. The server will only receive data.

Expected is that the network latency will not be to different from the initial measurement. Ethernet has duplex communication, therefore there should not be a lot of collisions occurring. The latency could however be increased due to the fact that the client needs to process all incoming packets.

B.3.6 A Setup Simulation

In this test a setup of the laboratory will be used as a reference for the amount of sensor signals and the amount of control signals which could go over the network. The setup chosen, is the youBot setup.

The youBot setup consists of nine motors, each motor has sensors for: position, velocity, current and temperature. In addition each motor can receive one control signal to control the motor. This results in a total of 36 sensor signals and 9 control signals.

Three scenarios will be measured:

Worst Case Scenario

Each signal is send as an individual packet with the same 128 byte size. The control signal witht $ID = 0$ will be used as the I/O control signal. The client will act as the “controller” sending 9 packets in the predefined frequency to the server. The server acts as the youBot target, it will send the 36 packets in the same frequency as the client to the client.

The expectation is that the performance will be normal for low network load. Increasing the frequency will increase the load, this should result in higher latencies.

Optimised Measurement 1: Fitting Data into Small Fixed Size Packets

In this test the signals are optimised, the packets size is kept at 128 bytes.. Each signal is considered to be of the `double` type, this an 8 byte data type. In order to identify each value send in a package, a header consisting of 2 bytes is added to the value, the first byte is a “tag” byte and the second a “size” byte. For each signal this means that it takes 10 bytes. Excluding the headers of Ethernet and UDP, the effective payload size for a 128 byte packet is 86 bytes.

This means that 8 values can be carried in each packet, resulting in two packets send from the controller and $36/8 = 5$ packets send from the plant. This results in lower traffic, it is expected that in these test the latency is decreased.

Optimised Measurement 2: Using Dynamic Packet Size

Ethernet allows for frames up to 1500 bytes, including all headers, this means that the payload is not limited to 86 bytes for the UDP/IP protocol. As with the previous test, each value is considered as a `double` , which is 8 bytes. Also the two identification bytes are included. The header of each frame is 42 bytes. For the controller this means that we have a packet of $42+90 = 132$ bytes and for the plant this means that in each cycle we send a packet of $42+360 = 402$ bytes.

This results in lower traffic. It is expected that in this test the latency is decreased in comparison with the previous test.

B.4 Measurement Results

The results of the measurements explained in section B.3 are presented here. In the measurements one Overo Fire executes the server application and will be referred to as the **server** in the results, the other Overo Fire executes the client application and will be referred to as the **client**. The PC will be referred to as the PC.

For the measurements both the maximum latency and the minimum latency are noted. Due to the fact that a scope is used to measure the latency, it is not possible to tell on a per packet basis what the latency is, therefore the mean of the latency is not regarded.

Because a non real-time Linux Operating System (OS) is used, the jitter of the 1000Hz loop in the client application was quite large, this made it difficult to perform the measurement. In order to do provide measurement results, the frequency for the I/O pin toggling was decreased.

Every 8 cycles the I/O pin was toggled, while the datagram packets were still send at the 1000Hz interval.

B.4.1 Initial Measurement Results

Table B.2 shows the results of the initial measurement. During one of the tests at 1000 Hz, a single spike during the test took 1960 μs. However this only occurred during one measurement and could not be reproduced.

Client frequency(Hz)	Min latency(μs)	Max latency(μs)	Load (packets/s)
100	267	370	100
1000	250	630	1000

Table B.2: The client sends 1 I/O control packet to the server in the given frequency.

B.4.2 High Traffic from the Client Results

Table B.3 shows the result of measurement. A note with the last measurement, packets displaying a high latency in the maximum range of 1990 μs was only a few per measurement and most packets stayed within a latency of 660 μs.

Client freq. (Hz)	Load freq. (Hz)	Load packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	100	100	265	428	10100
100	1000	100	265	468	100100
1000	100	100	246	744	11000
1000	1000	100	240	1990	101000

Table B.3: The client device sends 1 I/O control packet to the device server in the given frequency. On the client device the load application sends a burst of # packets in the given frequency to the PC.

B.4.3 Measurement High Data Traffic from Client to Server Results

Table B.4 shows the result of the measurement. In the tests where 100,000 packets/s were send to the server resulted in the server device displaying overloaded behaviour. The console via the serial interface was not responding, until the client was forced to stop sending packets.

Probably the amount of traffic causes the interrupt from the ethernet interface to stall all other processes on the system.

B.4.4 High Traffic to the Server from a Second Source Results

Table B.5 shows the results of the measurements.

When sending 100 packets at 100 Hz from the PC to the three different scenarios occurred depending on the start time of the application on the PC, resulting in different latencies. These three scenarios are shown in the table.

When sending 150 packets at 100 Hz, packet loss was registered on the server. Further increasing the amount of packets send by the load application resulted in a large packet loss. The I/O signal from the client device could not be identified at all at the server device.

Client freq. (Hz)	Load freq. (Hz)	Load packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	100	100	1200	1760	10100
100	1000	100	–	–	100100
1000	100	100	250	6600	10100
1000	1000	100	–	–	100100

Table B.4: The client device sends 1 I/O control packet to the device server in the given frequency. On the client device the load application sends a burst of # packets in the given frequency to the server device.

Client freq. (Hz)	PC freq. (Hz)	PC packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	100	10	262	400	1100
100	100	100	260	388	10100
100	100	100	890	1640	10100
100	100	100	4000	4700	10100
100	100	150	5800	7500	15100
100	100	250+	–	–	25100

Table B.5: The client device sends 1 I/O control packet to the server. The PC sends a burst of # packets in the given frequency to the server device.

B.4.5 High Traffic to the Client from the Network Results

Table B.6 shows the result of the measurement. The last measurement presented a problem observing the actual latency, due to the jitter in both the client application loop and the latency due to the collision it was not possible to read any latency on the scope.

B.4.6 Setup Simulation Results

Table B.7 shows the result of the **worst case** measurement. For the 1000 Hz frequency, the jitter of the loop application was to high in order to measure the network latency. Also when in the 1000 Hz the target application was started first, initially the control application registered missed packets, however at a certain point in time (after a few seconds) this behaviour settled down in no packet loss. However is the start order of the applications was reversed, the control application was started first, the target application registered packet loss. However in this case the target application did not settle to a no-packetloss state.

Table B.8 shows the measurement results of **optimised measurement 1**. At 1000 Hz, again the jitter of the main loop caused for unmeasurable scope results.

B.5 Result Analysis

In this section the results of the measurements are compared to the results which were expected.

Client freq. (Hz)	Load freq. (Hz)	Load packets	PC freq. (Hz)	PC packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	100	100	100	10	268	448	11100
100	100	100	100	100	270	614	20100
100	1000	100	100	10	258	466	101100
100	1000	100	100	100	268	472	110100
1000	100	100	100	10	250	830	12000
1000	100	100	100	100	250	530	21000
1000	1000	100	100	10	250	980	102000
1000	1000	100	100	100	–	–	111000

Table B.6: One client sends 1 packet to the server in the given frequency. On the client a load application sends a burst of # packets in the given frequency to the PC. The PC runs a similar load application, sending a burst of # packets in the given frequency.

Sample freq. (Hz)	# Sensor packets	# Control packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	36	9	270	1260	4500
1000	36	9	–	–	45000

Table B.7: Setup simulation measurement: **Worst case**. 36 sensor signals are send as individual 128 byte packets to the controller, 9 control signals are send as 128 byte packets to the target.

B.5.1 Initial Measurement

The expectation is that the latency would be higher than 100 μs and lower than 700 μs, is also the result of the measurement. The latency measured is between 250 and 630 μs.

With these latencies sample frequencies up to 500 Hz would be feasible. Considering a 1000 Hz sample frequency, with the maximum latency in mind, it would take two sample periods in order for the controller to respond to a measured signal.

B.5.2 High Traffic from the Client

The results that are expected are the same as with the previous measurement. Looking at the results, the measured latency is similar and stays within the 265 to 744 μs range.

The result in which both the load application and the client application

For the application, a non real-time Linux kernel is used, this means that both the server and client application did not get high priority and resource so they could meet their deadlines.

B.5.3 High Traffic to the Server from the Client

The expectation for this measurement is that they would be the same as the previous measurement. However this is not the case. The latency is quite higher than is the case in the previous measurement. When sending around 10,000 packets/s to the server the delay is measurable,

Sample freq. (Hz)	# Sensor packets	# Control packets	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	5	2	234	378	700
1000	5	2	–	–	7000

Table B.8: Setup simulation measurement: **Small packet optimisation**. 36 sensor signals are send as 8 byte double values with two byte identification headers in five 128 byte packets to the controller, 9 control signals are send as 8 byte double values with two byte identification headers in two 128 byte packets to the target.

Sample freq. (Hz)	# Sensor packet (bytes)	# Control packet (bytes)	Min latency (μs)	Max latency (μs)	Load (packets/s)
100	402	132	240	324	200
1000	402	132	234	1000+	2000

Table B.9: Setup simulation measurement: **Dynamic packet optimisation**. 36 sensor signals are send as 8 byte double values with two byte identification headers in one 402 byte packet to the controller, 9 control signals are send as 8 byte double values with two byte identifycation headers in one 128 byte packet to the target.

they delay is most likely caused because of the interrupts of the Ethernet card, interrupting the server application.

When sending a high amount of traffic, 100,000 packets/s the server did not update the I/O pin, showing an always high or low signal, depending on it's last state. In addition the screen session to the device also turned non responsive. This would suggest that the processing of the UDP datagram packets stall the server application.

B.5.4 High Traffic to the Server from a Second Source

In low network load, the latency is about the same as with the initial test. However on higher traffic load, the collisions within the network affect the performance. The latency is dependent on the distribution time in which the packets are send.

In the result three different scenarios occur. The highest delay most likely is the cause of the client and the PC sending their packets at the same time, this results in a high chance for collisions, causing high delays.

When the delays are low, this is most likely the case when the client and the PC send their data at different times in the in their 10 ms loop times.

B.5.5 High Traffic to the Client from the Network

The expectation for this measurement is that the latency is not affected much by the traffic send to the client. This is also the result of the measurement, the latencies that are measured are comparable to those of the initial test.

The latency of 980 μs is most likely due to the fact that the applications processing play a more important role in the delay.

B.5.6 A Setup Simulation

Three scenarios were measured. A worst case measurement and two optimised measurements. In all the measurements, the 1000 Hz sample frequency either should latencies which made it impossible to measure the latency with the scope in the described way. The last measurement showed measurable results for the minimum latency but not for the maximum latency.

Worst Case Scenario

In this measurement the minimum latency is as expected, however the maximum is quite high compared to the initial latency measurement. The cause could be either in the processing time, for sending the packets, or it could be caused due to the amount of time it takes to send the packets on the network.

Optimised Measurement 1: Fitting Data into Small Fixed Size Packets

The optimised measurement, decreases the load on the network. Also the amount of packets send is less, so the processing time for that decreases. The result is that we obtain latencies which are about equal that that of the initial measurement.

Optimised Measurement 2: Using Dynamic Packet Size

The second optimised version also decreases the network traffic, but now by increasing the packet size. This results in a slight improvement of the maximum latency in the 100 Hz test and also showed improvement in the measurement of the 1000 Hz signal.

The result is most likely due to two things:

1. Only one packet has to be created and filled with data, this saves processing time of creating individual packets.

2. Because only one packet has to be send, the additional overhead of ethernet headers and additional bytes, like the ethernet preamble, don't have to be send over the network.

B.6 Conclusions

B.6.1 Latency and Sample Frequency

In the initial test we see that we have at least a latency of 250 μs up to 630 μs. This means that we already have constraints on the maximum sample frequency we can achieve when network communication is included within the control loop.

In the initial measurement, without taking into account additional time which is needed to calculate the control value, the controller can respond to a measured signal in about 1.26 milliseconds. For low frequency control loops, 500 Hz and lower, this should not cause an issue. When using faster loops, for example 1000 Hz, the delay in the control signal could become a problem.

When applying duplex communication, results from B.4.6, we see that this impacts the performance when the amount of packets send is high. Therefore the amount of traffic which can be send at a high sample frequency is limited.

B.6.2 Applications

Ethernet could be used in order to connect devices, for which slow sample frequencies for the data are required. In all the measurements, the latency varied from 250 μs up to a few milliseconds. For frequencies of 100 Hz and slower this is a feasible transport possibility.

However for faster sample frequencies measures need to be taken, or different communication platforms should be used, that do offer the fast transmission of data.

B.7 Recommendations

B.7.1 Software and Hardware Limitations

In the measurements the used platform is Linux, this is a non real-time OS. The processing time, considered in the latency could be improved by using a real-time operating system. However using a real-time operating system is not a guarantee that the performance is improved, this should be measured in order to validate this.

B.7.2 Transmission protocol

Another aspect which could improve the latency is the protocol used. Now the UPD/IP protocol is used, this means that on the processing size, two additional network layers have to be processed: UDP and IP. Using raw Ethernet packets could also result in an improvement in the latency between the two devices.

B.7.3 Topology

Another aspect which could improve the latency is the network topology. In the measurement a switch is used in order to connect the two devices. However using a direct connection, between the two devices could speed up the the performance of the network as the packets don't need to pass through the switch device.

However how large the effect of the switch is should be measured first by creating a setup with a direct connection.

C Real-time Ethernet Protocols

C.1 Introduction

Ethernet is one of the most common forms of network communication. It is very easy to setup and very cheap hardware is available for the technology. Due to the implementation of Ethernet, it is not a real-time transmission protocol. This is due to the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) mechanism, which makes sure that on the same line two nodes are not sending at the same time. This mechanism causes for non deterministic latencies on the network.

There are several projects running which try to address the this issue of non deterministic delays and use Ethernet as a real-time network. The website Laboratory of Process Data Processing of Reutlingen University (2013) gives an overview of solutions used in the automation industry. The list contains both commercial projects and open (source) projects. This appendix highlights a few of the open protocols available in the list. Briefly the protocol is explained in how they address the delay issues with Ethernet and what is required in order to use the protocols.

The following protocols are highlighted:

- EtherCAT
- Real-Time Publisher Subscriber protocol
- Ethernet POWERLINK
- RTnet

C.2 Ethercat

C.2.1 About EtherCAT

The EtherCAT protocol is maintained by the EtherCAT Technology Group (ETG). The technology/protocol is an open protocol, however there is the requirement that you are a member of the ETG. The protocol is based on the CANOpen protocol and functions as a bus protocol (Ethercat Technology Group (ETG), 2012).

C.2.2 Technology Overview

The EtherCAT protocol uses a bus principle in order to send data to the various nodes in the network. A master node in the network commences a transmission cycle by sending an EtherCAT packet on the network to the first node in the cycle of that packet. Each node processes the packet “on the fly” and then passes it on to the next node in the cycle. The last node sends the packet back to the master. Due to the duplex nature of Ethernet, this means that for the packet there are no connections as the packet only travels one way in the network.

Each slave node in the network does the processing on the packet in the Ethernet hardware. Data exchange between the Ethernet hardware and processing unit of the slave is done via a shared memory. This means that the processing unit is is solely responsible for processing the Ethernet packet and that the processing unit is not required to spend resources on this. Due to this, the delay in the network is only caused by the hardware delay due to transmitting the packet through the network and the processing of the packet in the Ethernet hardware.

The EtherCAT protocol itself is mapped directly onto the Ethernet protocol. This means that no additional processing is required for higher protocol layers and that there is more space for data in the packet, as Ethernet frames have a limited size.

C.2.3 Requirements

Due to the fact that processing of the Ethernet packets is done in the Ethernet hardware, special Ethernet hardware is required for this and conventional Ethernet hardware cannot be used. In addition, to prevent collisions only nodes supporting the EtherCAT protocol should be included in the network, this means that a dedicated network is required for EtherCAT.

C.3 Real-Time Ethernet

C.3.1 About RTnet

Real-Time Ethernet (RTnet) is developed and maintained by the Institute of Systems Engineering, Real-Time Systems Group. It is now an open source project, that also has contributors from all over the world. The protocol is an open protocol and implementations are available for the linux platform. The protocol uses time slicing of the network time in order to deal with the unpredictable delays in Ethernet.

C.3.2 Technology Overview

The protocol uses time slices in order to regulate network traffic. Each time slice begins with a synchronisation frame from the master. These are followed by synchronisation frames from backup masters, should the main master node fail. Each slave node then has a fixed amount of time to send their data on the network, this time is defined in a schedule.

In order to implement the protocol the Ethernet stack of the Operating System (OS) is replaced by the RTnet stack. The stack takes care of the protocol implementation, in addition it also replaces the default OS stack with a deterministic network stack. Here, indeterministic memory allocations are replaced with preallocated buffers.

C.3.3 Requirements

RTnet is a software implementation, it replaces the network stack of the operating system with the RTnet stack. In addition it uses time slices in order regulate Ethernet traffic. The network needs to be a private network containing only nodes which implement the RTnet protocol.

C.4 Ethernet POWERLINK

C.4.1 About Ethernet POWERLINK

Ethernet POWERLINK is a real-time Ethernet protocol maintained by the Ethernet POWERLINK Standardisation Group (EPSG). It is an open protocol. Ethernet POWERLINK implements time slice principles in order to prevent collisions, which have an undefined delay within Ethernet.

C.4.2 Technology Overview

Ethernet POWERLINK replaces the current Ethernet stack to implement the protocol. The network time is divided into slices. Each time slice starts with a Start of Cycle synchronisation frame. After this, the master polls each slave node. The slave node then gets a fixed amount of time to send its data on the network. Each time slice ends with a limited time to send non real-time communication.

Giving slave nodes fixed time slots to send their data, makes sure that collisions do not happen within the network. The collisions cause unpredictable behaviour in the delay on the network.

C.4.3 Requirements

The devices in the network require special drivers in order to implement the time slice principle, special hardware is not required as the protocol is implemented in software. In order to have the network real-time, only nodes implementing Ethernet POWERLINK should be connected to the network, thus it requires a private network.

C.5 Real-Time Publisher Subscriber Protocol

C.5.1 About Real-Time Publisher Subscriber Protocol

The Real-Time Publisher Subscriber (RTPS) protocol is maintained by the Object Model Group (OMG). The specification is open and can be obtained from the OMG website. The RTPS protocol is a platform specific implementation of the Data-Distribution Service (DDS) specification.

C.5.2 Technology Overview

The protocol is based on the publisher-subscriber principle. Data producers publish data, sources that require the data can subscribe to the producing source. The protocol introduces mechanics to send and receive data between nodes and includes Quality of Service (QoS) services for information about the network.

The protocol is not a real-time protocol, in order to minimise latency, network traffic should be kept low. Therefore a dedicated network should be used that is separated from busy networks, like the Internet.

C.5.3 Requirements

The protocol does not require specific hardware. The various services described in the specification should be implemented in a server, or daemon which responsible for the network communication. Although there are no requirements for the network, a low traffic dedicated network is preferred.

C.6 Conclusion

Several real-time Ethernet protocols, used in industry have been highlighted in this appendix. They all use different approaches to deal with the indeterministic delay behaviour of Ethernet, so it is applicable for real-time communication.

Bibliography

Arduino (2013), Arduino.
http://arduino.cc

Bennett S. (1988), *Real-Time computer control: An introduction*, NY: Prentice-Hall.

Bezemer, M. M. (2013), *Cyber-Physical Systems Software Development - way of working and tool suite*, Ph.D. thesis, University of Twente, doi:10.3990/1.9789036518796.

Broenink, J. F., M. A. Groothuis, P. M. Visser and M. M. Bezemer (2010), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, Eds. D. Kubus, K. Nilsson and R. S. Johansson, IEEE, IEEE, pp. 73 – 77.

Controllab (2013), Controllab.
http://www.controllab.nl

Dolejs, O., P. Smolik and Z. Hanzalek (2004), On the Ethernet use for real-time publish-subscribe based applications, in *IEEE International Workshop on Factory Communication Systems*, IEEE, pp. 39–44.

Elmo Motion Control (2013), Elmo Motion Control.
http://www.elmomc.com

Ethercat Technology Group (ETG) (2012), *EtherCAT - the Ethernet Fieldbus*, Ethercat Technology Group (ETG).

Franken, M. C. J. (2011), *Control of haptic interaction : an energy-based approach*, Ph.D. thesis, Univ. of Twente, Enschede.

Geus, W. d. (2012), *Design and Realisation of a Biped Walker with Compliant Legs*, Msc report 012ce2012, University of Twente.

Groothuis, S. S. (2011), *Design, Modeling and Control of a Rotational Variable Stiffness Actuator*, Msc report 020ce2011, University of Twente.

Gumstix (2013), Gumstix.
https://www.gumstix.com

Ketelaar, J. G. (2012), *Controller design for a Bipedal Robot with Variable Stiffness Actuators*, Msc report 035ram2012, University of Twente.

Khalil, I. S. M., R. M. P. Metz, B. A. Reefman and S. Misra (2013), Magnetic-based minimum input motion control of paramagnetic microparticles in three-dimensional space, in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 2053 – 2058.

Laboratory of Process Data Processing of Reutlingen University (2013), Information about Real-Time Ethernet in Industry Automation.
http://www.real-time-ethernet.de

Lev Walkin (2013), ASN.1 Compiler.
http://lionet.info/asn1c

Mathworks (2013), Mathworks.
http://www.mathworks.com

Michael, M. M. and M. L. Scott (1996), Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, ACM, pp. 267–275.

NXP (2013), MBED.
http://www.mbed.org

Telecommunication Standardization Sector of ITU (2002a), *Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*, International Telecommunication Union (ITU).

Telecommunication Standardization Sector of ITU (2002b), *Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, International Telecommunication Union (ITU).

The Open Group (2013), The Open Group Base Specifications Issue 7.
http://pubs.opengroup.org/onlinepubs/9699919799/

The World Wide Web Consortium (W3C) (2005), XML Schema.
http://www.w3.org/2001/XMLSchema

Vrooijink, G. J., M. Abayazid and S. Misra (2013), Real-time three-dimensional flexible needle tracking using two-dimensional ultrasound, in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, IEEE.

Wilterdink, R. J. W. (2011), *Design of a hard real-time, multi-threaded and CSP-capable execution framework*, Msc thesis 009ce2011, University of Twente.

Xenomai (2013), Xenomai.
http://www.xenomai.org