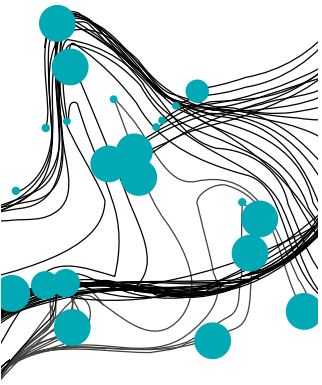
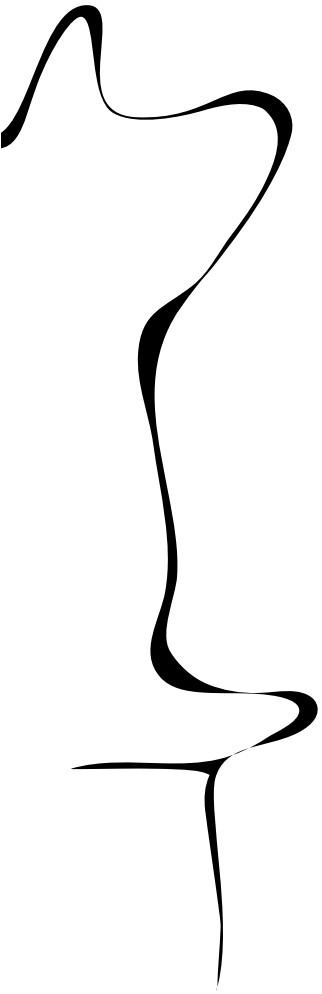


RAM ROBOTICS AND MECHATRONICS



INTEGRATION OF HARD AND SOFT REAL-TIME TASKS IN CYBER-PHYSICAL SYSTEMS

A. (Arnold) Hofstede

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
prof. dr. ir. G.J. Heijenk

September, 2020

045RaM2020
Robotics and
Mechatronics EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

Cyber-Physical Systems have become more and more complex, increasing the amount of required computational resources. On battery powered devices, such as drones and mobile robots, computational resources are limited. Moving the execution of resource intensive algorithms that have no strict timing requirements from a resource-constrained embedded device to a resource-rich system is a solution for this. This method is used at RaM in the model-driven design tool-chain TERRA by connecting ROS to LUNA.

A bridge from ROS to LUNA exists, but has limitations that causes that the bridge is not used. In this project the usability of the ROS-LUNA bridge in distributed Cyber-Physical Systems is improved and means to integrate soft real-time tasks in the design process of Cyber-Physical Systems are provided.

To improve the ROS-LUNA bridge an analysis on the existing implementation is done. This analysis showed that the implementation severely limits the throughput of the bridge, causing a loss of data when transmitting large amounts of data. The use of TCP for communication between the ROS side and LUNA side of the bridge causes latency spikes when packets are lost. In addition, signals transmitted over the ROS-LUNA bridge have a latency that is a multiple of the network delay when periodic data is transmitted due to Nagle's algorithm.

The implementation of the bridge in ROS is refined such that it decodes all values in the incoming packets immediately, resulting in an increased throughput. By adding the option to use UDP as transport protocol and to disable Nagle's algorithm, the usability of the bridge is improved on systems that are connected by an imperfect network. An evaluation of the bridge using UDP as a transport protocol shows that latency spikes no longer occur due to lost packets. When using UDP or TCP with Nagle's algorithm disabled the latency of the signal is equal to the delay on the network link.

Several CSP patterns are presented to connect SRT and HRT tasks in applications that are developed using the TERRA/LUNA tool-chain and ROS. The patterns to decouple HRT from SRT process execution are evaluated on a stressed real-time system that uses the Xenomai co-kernel. The measurements on patterns used to transmit data from LUNA to ROS show that there is no effect on the amount of jitter in the HRT process when a naive implementation to connect outgoing ROS channel is used. The measurements on patterns used to receive data from ROS in LUNA show that naive implementations to connect incoming ROS channels to LUNA result in synchronization between the HRT process and the rate at which messages are received. The insignificant difference in results between using proper coupling patterns or naive connections can be explained by to the abundance of computational resources available on the measurement setup.

Due to covid-19 restrictions, an evaluation of the patterns to connect SRT and HRT tasks could not be performed on a resource-constrained embedded device. In addition, an implementation of embedded control software using the improved ROS-LUNA bridge for a representative demonstration setup could not be achieved.

Contents

1 Introduction

1.1 Context

A Cyber-Physical System (CPS) consist of a physical plant that is controlled by collaborating computational algorithms. CPS are emerged from traditional embedded systems into a network of interacting computational elements with physical inputs and outputs. Due to CPS becoming more and more complex, the amount of required computational resources increases. On battery powered devices, such as drones and mobile robots, computational resources are limited.

The algorithms for interaction with the physical plant, such as loop controllers and safety layers, typically have hard real-time (HRT) requirements since the system must be reliable and safe. Higher-level controllers, such as sequence and supervisory controllers, do not have such strict real-time requirements and are classified as soft real-time (SRT). Missing a deadline in these algorithms only reduces the usefulness of the result.

Distribution of software over multiple computational devices is a solution to reduce the need for computational resources on the embedded system when system costs must be reduced or the energy budget is limited. Resource intensive tasks without HRT requirements can be offloaded to a remote resource-rich platform. For communication with these tasks it is possible to use conventional (wireless) networks.

At the Robotics and Mechatronics (RaM) group at the University of Twente, a model-driven tool-chain TERRA (Twente Embedded Real-time Robot Application) is developed to aid in the design of HRT embedded control software that uses LUNA (Luna Universal Network Architecture) as a middleware (Bezemer, 2013). An extension to connect LUNA to ROS (Robotic Operating System) is made by Werff (2016) to offload SRT tasks to a remote platform.

A CPS considered in the context of this thesis consists of a resource-rich platform that is connected to an embedded system by an imperfect network. On the embedded system an HRT LUNA control application is executed that interacts with the physical system and with algorithms on a resource-rich platform over the network. In addition, other devices such as a camera can capture the physical system and provide data to the SRT algorithms over the network. A diagram of such a system is given in Figure 1.1.

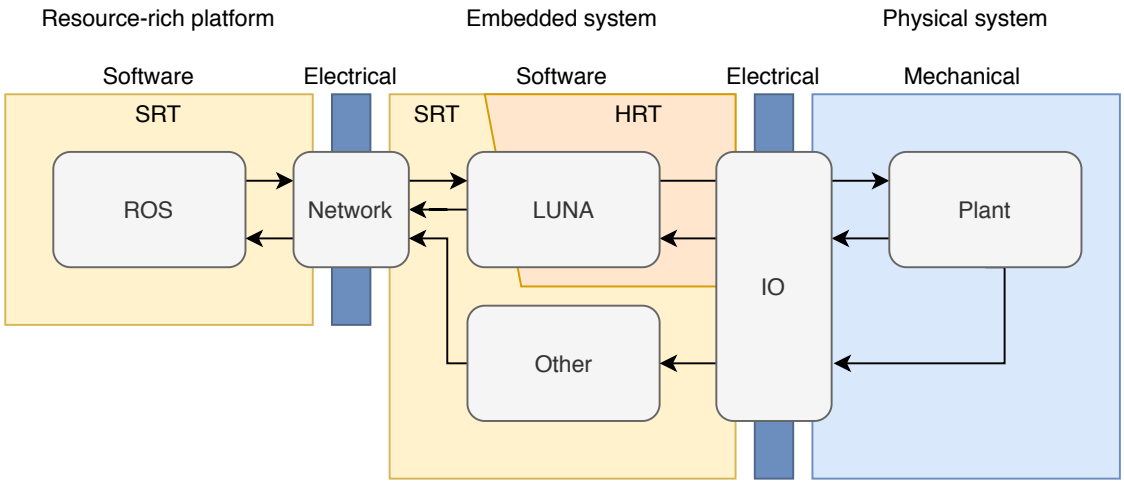


Figure 1.1: Typical Cyber-Physical System considered in this thesis.

Even though a connection from LUNA to ROS exists since 2015 (Vos, 2015), it is not often used. An platform analysis done on projects within the RaM group by Kempenaar (2014) concludes that loop control is typically not part of the research, resulting in the use of ready-to-use off-the-shelf controller components. In some of the control setups, the control algorithms were not limited to dynamic equations. Image processing and path planning provided by a third party were also part of the control software, adding additional requirements on the development tool.

An early version of the ROS-LUNA bridge is used by Vos (2015) to add a vision-based sorter module to a production cell setup. Ridder (2018) used the same setup for a use-case driven analysis of the TERRA tool-chain, but used native ROS-channels since the limitations of the ROS-LUNA bridge made it an unsuitable option. The existence of these limitations are in conflict with the design philosophy of the bridge that is to be used without modifications in all situations.

1.2 Project goals

The goals of this project concern the usability of the ROS-LUNA bridge and integration of soft and hard real-time software in a broader context. These goals are formulated as follows:

1. *Improve usability of the ROS-LUNA bridge in distributed Cyber-Physical Systems.*
2. *Provide means to integrate soft real-time and hard real-time software in the design process of Cyber-Physical Systems.*

1.3 Thesis outline

In Chapter 2, background material on real-time, the RaM ECS tool-chain, ROS and the ROS-LUNA bridge is given. In Chapter 3, an analysis on a typical system, the ROS-LUNA bridge and connections between SRT and HRT software is presented, resulting in a list of project requirements. In Chapter 4, the refinements to the ROS-LUNA bridge are given and discussed. In Chapter 5, a generic and TERRA-specific approach to coupling SRT and HRT tasks is given and discussed. Chapter 6 ends with conclusions and recommendations.

2 Background

In this chapter, background information relevant to this thesis is given. In section 2.1, information on real time and the used real-time operating system are presented. Next, in section 2.2, the RaM ECS tool-chain consisting of TERRA and LUNA is discussed. In section 2.3, information on ROS is given. In section 2.4, the existing ROS-LUNA bridge implementation is discussed. Lastly, in section 2.5, relevant transport types and Nagle's algorithm are presented.

2.1 On real time

The different notions of real time considered during this thesis are soft, firm and hard real-time. Below properties and examples of the types of real-time are given.

- **Hard real-time (HRT):** No deadlines can be missed without breaking the system. An example of such a system is an airbag control system that needs to respond within a microsecond range. HRT systems such as a missile defense system also needs to responds before a deadline, but response times are not as fast.
- **Firm real-time (FRT):** Some deadlines can be missed without breaking the system, but results are not longer useful. A loop controller is an example of such a system. Missing a single deadline does not have catastrophic consequences. However, it is hard to define the number of accepted missed deadlines and to design a system that does not violate those requirements.
- **Soft real-time (SRT):** Deadlines can be missed without breaking the system, but results are less useful. Examples of these systems are path planning and image processing algorithms. These algorithms are typically resource intensive and only degrade the quality of service of the system when missing a deadline.

Utility functions that represent the effect of missing a deadline in a computation for different types of real-time systems are given in Figure 2.1.

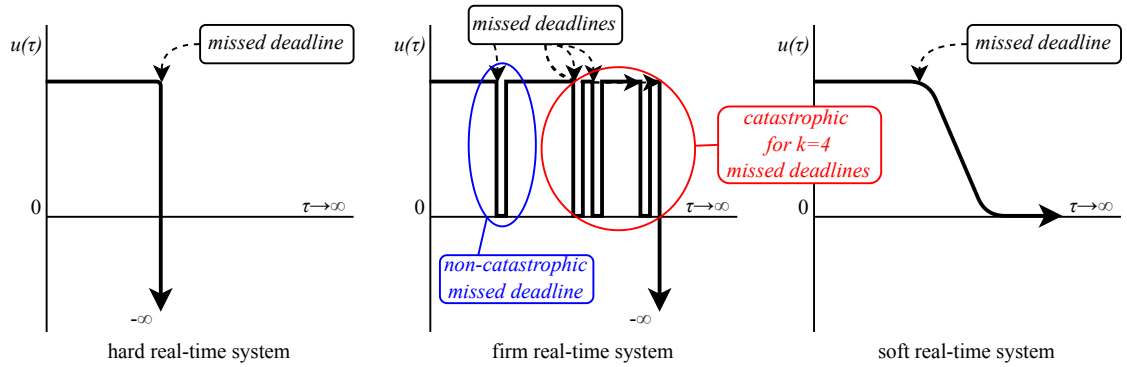


Figure 2.1: Utility functions $u(\tau)$ for different notions of real-time when missing a deadline as given by Boode (2018).

The layered approach for software architecture for embedded systems in Figure 2.2 allows for the different levels of real time. In this approach, the loop controllers are classified as HRT. This classification is also used in this thesis. In the layered software architecture, the ROS-LUNA bridge is located on the *Sequence control* layer in the SRT domain.

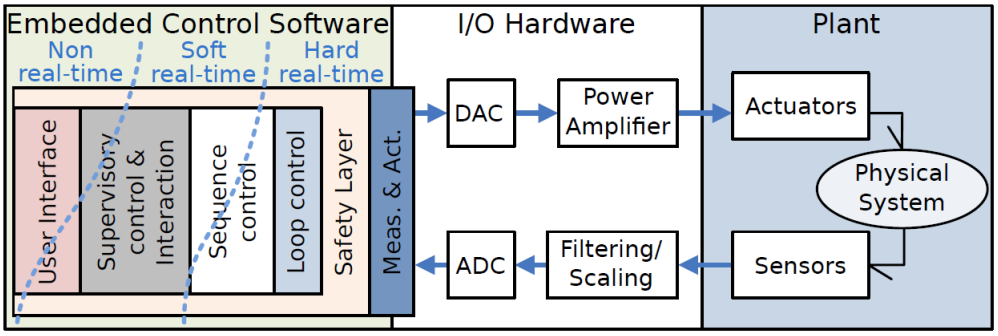


Figure 2.2: Layered software architecture for embedded systems (Broenink et al., 2010).

Normal operating systems such as Linux are not capable to guarantee meeting deadlines when executing real-time software. The Xenomai 3.1 co-kernel[1] is used in this thesis to meet real-time requirements. This target OS is supported for applications that are developed using the RaM ECS tool-chain on multiple hardware architectures.

There are two options to let Xenomai deliver real-time. One is by supplementing Linux with a real-time co-kernel, named Cobalt, running side by side it. The other option is by relying on the real-time capabilities of the native Linux kernel, forming the Mercury core. In Figure 2.3, a diagram representing the Xenomai dual kernel configuration is given. During this thesis the dual kernel option is used.

The POSIX API provided by Xenomai is used in LUNA as a interface with the Cobalt core. Since this API is only a subset of the complete POSIX API some functions are only provided by the Linux API. An Xenomai application can run in two modes: primary mode, where it is scheduled by the Xenomai kernel, and benefits from hard real-time scheduling latency, or in secondary mode, where it is an ordinary Linux thread and may call any Linux services. A real-time application can make a call to a function that is only provided by the Linux API, but in that case it switches to secondary domain. This results in the task being scheduled by the Linux scheduler and HRT guarantees are lost.

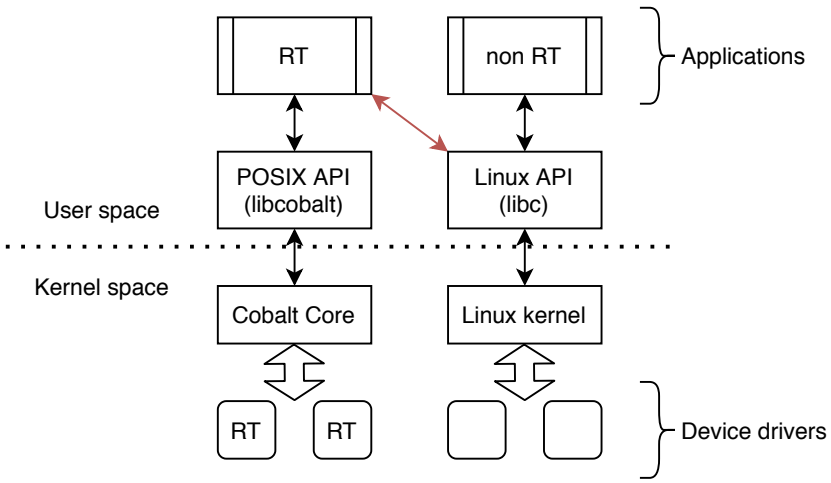


Figure 2.3: Xenomai dual kernel configuration (Xenomai, 2020).

2.2 RaM ECS tool-chain

The ECS tool-chain (Bezemer, 2013) consists of TERRA and LUNA, and allows developers of CPS to create control software using model-driving engineering. Control algorithms can be

[1] https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home

imported from 20sim and using FMU's it is possible to simulate the developed software with models from tools that support this standard. In the tool-chain, Communicating Sequential Processes (CSP) (Hoare, 1978) is used to formally describe software architectures.

2.2.1 LUNA

LUNA is a C++ library that is hard real-time, multi platform, scalable, component based and features a CSP execution engine. The code generated in TERRA depends on this library. Since the library is multi platform, it is possible to develop applications from the same model for Linux or Linux with the Xenomai co-kernel on either x86/x64 or armv7 depending on configuration.

Communication between CSP processes is done using rendezvous channels. Rendezvous communication requires both reader and writer to be ready before transferring data. Non-blocking buffered channels are available for processes that do not run on the same frequency. When using these channel deadlock does never occur.

The hard real-time properties of LUNA applications are only guaranteed when executed on a properly configured system. The execution component in LUNA determines the application flow in accordance with the CSP model made in TERRA. LUNA provides Run-to-Completion scheduling for CSP processes by the CSP execution engine as is prescribed by CSP.

2.2.2 TERRA

TERRA is a model-driven design tool that supports both architecture and CSP modeling. On the architecture level the physical system can be modeled, existing of hardware and software interfaces, software (CSP) models and models of the plant when simulation is desired. On the CSP level a model of the embedded software can be made.

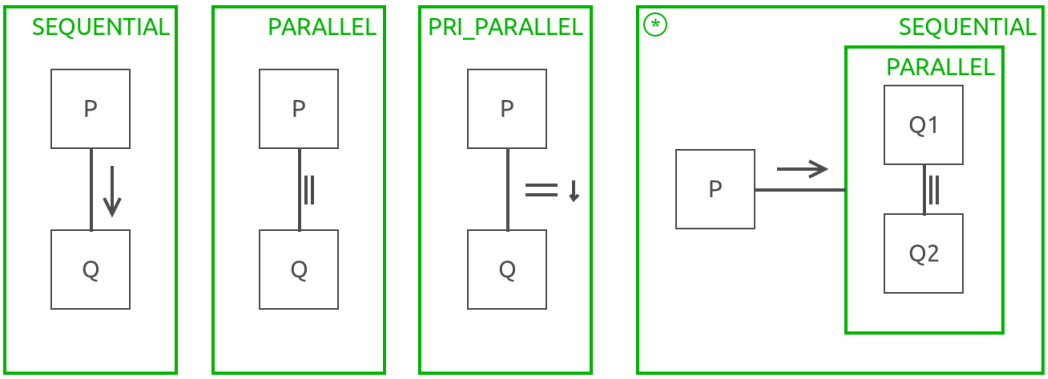


Figure 2.4: CSP constructs in TERRA used in this thesis.

CSP models are modeled using gCSP by Bezemer (2013). 20-sim and FMU interface models can be converted by TERRA into CSP models such that they can be used in LUNA applications. In Figure 2.4, the constructs in TERRA relevant to this thesis are given. Additional information on these constructs is given below:

- **Sequential:** Executes processes sequentially.
- **Parallel:** Executes processes in parallel and gives each process a fair chance of being executed first.
- **Pri-parallel:** Executes processes in parallel and gives each process a fair chance of being executed first. Different OS scheduler priorities are assigned to the individual processes. The priorities only effect execution when both parallel processes are recursive.

- **Group:** Groups processes by their compositional relation to others. In TERRA groups are modeled by green squares.
- **Recursion:** Starts execution of processes inside a group when all processes in the group are finished. Recursion is indicated by a encircled asterix in the top-left corner of a process or group.

In CSP, processes communicate by using channels. In TERRA these channels are represented by an arrow that is either filled in or empty depending on whether a buffer is used or not. In TERRA there is no way to indicate visually if the channel is blocking or non-blocking. The arrow used for a buffered channel is given in Figure 2.5a and the arrow used for a unbuffered channel is given in Figure 2.5b.

(a) Buffered **(b)** Unbuffered

Figure 2.5: CSP channels in TERRA.

Ports on the architecture level represent links to either hardware or other software. These are connected to Writers or Readers inside a CSP model and have the same functions as CSP channels. The code of a hardware channel interacts with real hardware and is executed within the same thread as the Writer or Reader. This is also the case for software channels. Timer ports are available on the architecture level. A writer in a CSP model that writes to this port blocks execution of the model until the timer ticks.

In the code that is generated by TERRA all CSP processes are mapped to threads, without differentiation between real-time and non real-time threads. With the use of Xenomai's POSIX skin, all threads are default scheduled as real-time when compiling the application for Xenomai. Assigning priority to differentiate between the critical and less critical threads in TERRA is also not possible.

2.3 ROS

Robotic Operating System (ROS) is robotic middleware consisting of libraries and tools to help create robot applications. ROS has become the defacto standard robotics middleware for hobbyists and academics. In addition to the benefit of only having to focus on the algorithm that is part of the research, ROS also enables users to share their work.

ROS and LUNA are both middleware that provide composition, coordination and communication in software. ROS is not designed with real-time as a goal, but it is possible implement a hard real-time controller within a ROS node by decoupling HRT code from ROS code and running a real-time OS on the target system. Communication between ROS nodes is done using publish-subscribe mechanisms.

2.4 ROS-LUNA bridge

The ROS-LUNA bridge by Werff (2016) consists of three parts: the ROS side of the bridge, the LUNA side of the bridge and a network in between. Communication between the ROS and LUNA side is done over TCP/IP. A native ROS port is added to LUNA by Ridder (2018), but requires ROS to be installed on the target platform. In addition, only single value messages are supported by this native implementation.

The ROS side of the bridge is designed such that multiple LUNA clients can connect. After connection, topic connect packages are sent to ROS and the topic listeners and run-time binding publishers are configured. When a topic listener receives a message, a single TCP/IP packet is

send to the LUNA side of the bridge. When the ROS side of the bridge receives a TCP/IP packet it is decoded and published to a topic using the corresponding run-time binding publisher(s).

Interaction of the CSP model with the ROS-LUNA bridge depends on the configuration of the bridge ports on the architectural level. Readers can read from the ROS channel similarly as from a blocking or non-blocking buffered CSP channel. Writers can either write to the channel and directly send the value in a TCP/IP packet to the ROS side of the bridge or store the value in a buffer. When the value is stored in a buffer, the call required for sending the TCP/IP must be done somewhere in code.

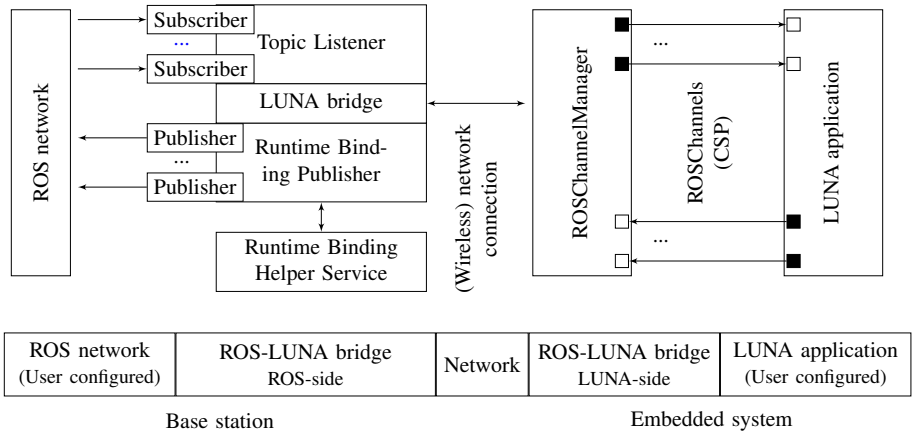


Figure 2.6: Block diagram of ROS-LUNA bridge (Werff, 2016).

2.5 Network

In the existing ROS-LUNA bridge, TCP is used as transport protocol. TCP provides reliable, ordered and error-checked delivery of a stream of bytes between applications running on hosts communicating via an IP network. Another transport protocol is UDP, which is connection-less and has a minimum of protocol mechanisms. Due to the lack of protocol mechanisms such as retransmission of lost packets, UDP is preferred in time-sensitive applications. In Table 2.1 properties of TCP and UDP are given.

Table 2.1: Properties of TCP and UDP.

TCP	UDP
Connection oriented	Transaction oriented
Retransmission of lost packets	No retransmission of lost packets
Ordered data transfer	
Flow control	
Congestion control	

In the TCP protocol, Nagle's algorithm is enabled by default. This algorithm is defined by the RFC[2] as follows: *"Inhibit the sending of new TCP segments when new outgoing data arrives from the user if any previously transmitted data on the connection remains unacknowledged."*.

In Figure 2.7 the effects of Nagle's algorithm are sketched. Data value 2 is stored in a buffer until the acknowledgment of packet 1 is received. When the acknowledgment for packet 2 is arrived, both packet 3 and 4 are transmitted. Waiting for the acknowledgment results in an increasing latency until multiple packets can be sent together.

[2]RFC 896: https://tools.ietf.org/html/rfc896

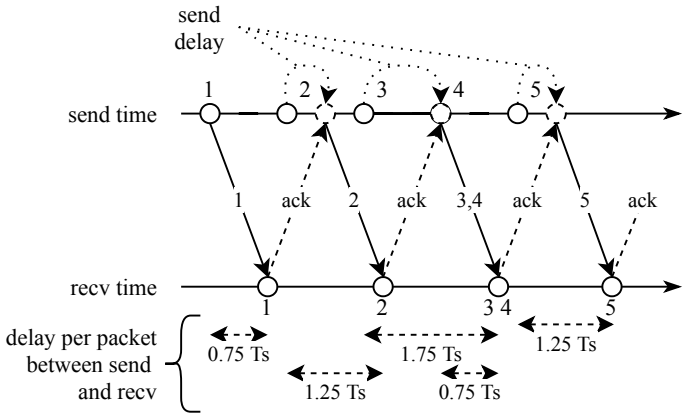


Figure 2.7: Sketched effects of network delay on latency due to Nagle's algorithm.

3 Analysis

In this chapter an analysis is done to determine the requirements necessary to achieve the goals stated in section 1.2. Firstly, in section 3.1, typical systems and their challenges in connecting tasks with various real-time requirements are discussed. Next, in section 3.2, advantages and disadvantages of using the ROS-LUNA bridge or native ROS implementation are discussed. In section 3.3, the effects of delay and packet loss in imperfect networks on periodic signals are discussed. In section 3.4, differences between ROS and CSP channels, and real-time requirements are discussed. Finally, the requirements following from this analysis are discussed.

3.1 Typical system overview

The typical system considered in this thesis that uses the ROS-LUNA bridge is drawn in Figure 1.1 and described in the introduction as an embedded device with a LUNA application that interfaces with a physical plant and is connected by a network to a resource-rich platform with a ROS application that performs computation-intensive algorithms. In this section the needs with respect to connections between subsystems are further analyzed.

The Production Cell and JIWY setup are often used in the RaM laboratory as a CSP research setup. The Production Cell setup represents an injection molding machine and has six degrees of freedom. The JIWY setup is smaller and consists of a camera that can tilt and pan. This setup is used by Werff (2016) to show the correct working and functioning of the earlier version of the ROS-LUNA bridge and will also be considered in this analysis. The goal of this setup is to track an object by following set-points that are generated with an image processing algorithm on a remote platform. A top-level architecture of this system is given in Figure 3.1 which describes the distribution of ECS layers from Figure 2.2 over the devices. Since communication between LUNA and ROS occurs in this system on the sequence layer, this layer is represented on both devices.

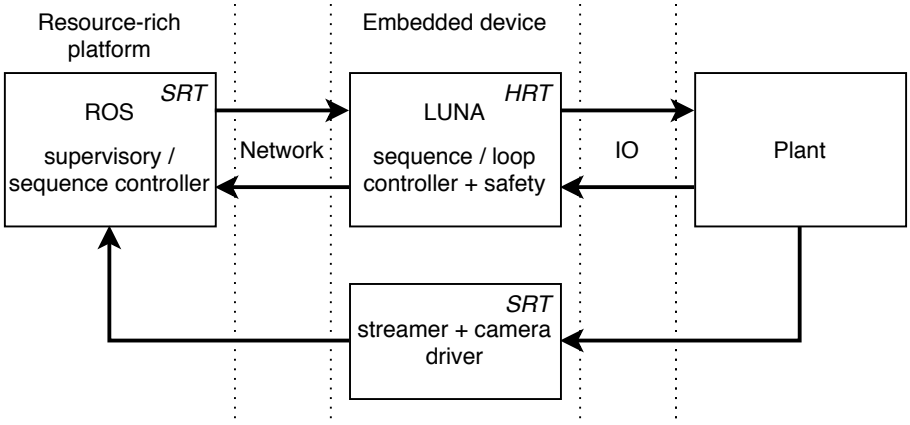


Figure 3.1: Top level architecture of the vision based JIWY setup.

In Figure 3.2 an embedded control software architecture for the JIWY setup in TERRA is given. In addition to executing the control software, a camera stream is send from the embedded device to the resource-rich platform. The control software receives periodic set-points at the frequency at which the camera records frames and sends its current pan and tilt angle to ROS for visualization purposes. A status variable is communicated to ROS when the system changes state.

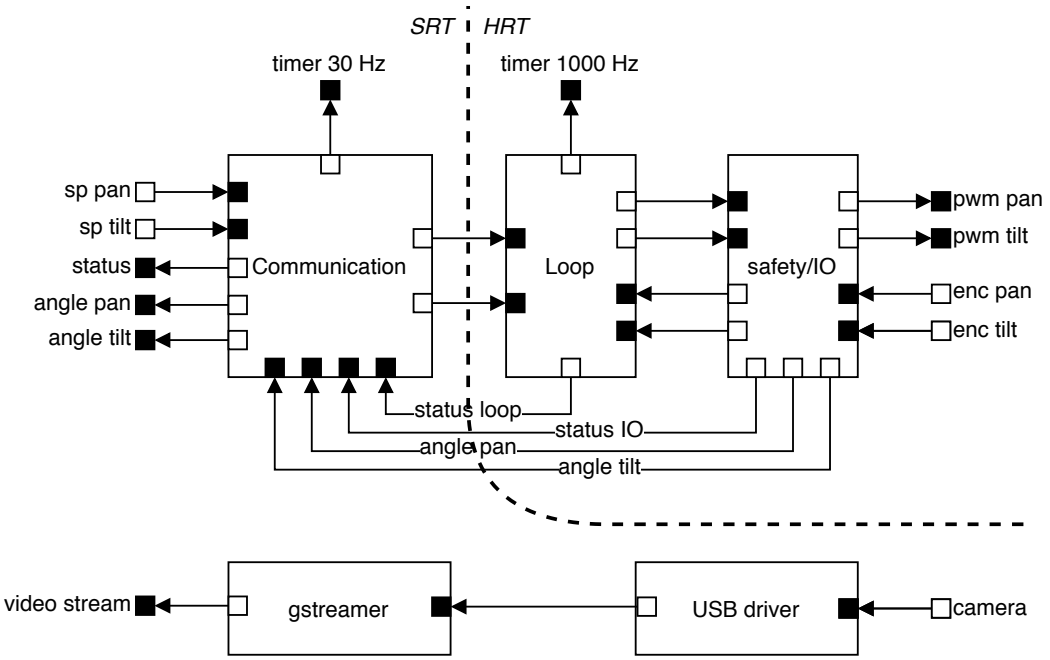


Figure 3.2: Embedded control software architecture for the JIWY setup.

During the design of such a system, several challenges with respect to coupling SRT and HRT processes can be observed. The main challenge is to guarantee execution of *Loop* and *Safety/IO* within a specified time frame while interacting with *Communication* that has SRT requirements. Another challenge is maximize the utility of signals send from ROS to LUNA and vice versa.

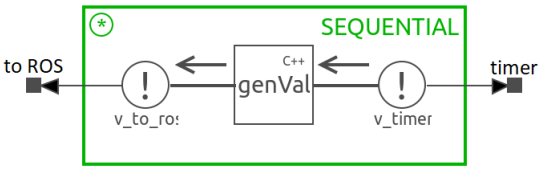
For a HRT subsystem, all in- and outgoing channels to SRT subsystems need to be buffered and non-blocking to guarantee meeting real-time requirements. Hardware ports for encoders and PWM signals need to be HRT by design since they are in the HRT control loop. Network ports within the scope of this thesis cannot guarantee to meet these requirements and must therefore be kept outside the HRT subsystem.

The means to achieve the highest utility varies for different types of signals. For a system that tracks set-points, the utility of a value is highest when the latency is minimal and decreases to zero when a newer value is arrived. For these signals, loss of a value has less impact on utility than the increased delay introduced by re-transmitting a value. For a system that moves from set-point to set-point in order to follow a trajectory, the priority shifts to reliability. Sporadic data such as status information also prioritizes reliability since the re-transmission time is typically less than the time a next value is send.

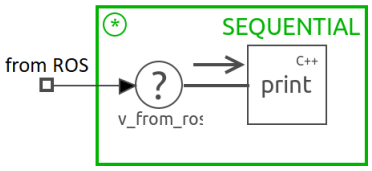
3.2 Bridge or native ports

Two ways of connecting LUNA to ROS exist, the ROS-LUNA bridge port by Werff (2016) and the native ROS port by Ridder (2018). Both the ROS-LUNA bridge port and native ROS port to connect ROS and LUNA have their advantages and disadvantages. These are further analyzed in the subsections below.

In addition, the throughput of the original existing ROS ports is measured using the LUNA applications in Figure 3.3. In ROS, command line tools are used for publishing and subscribing to the topic used for communication. The frequency of sending values from LUNA to ROS is determined by a timer port in LUNA, whereas the frequency of sending values from ROS to LUNA is determined by the frequency of the publisher in ROS.



(a) CSP model used for sending values.



(b) CSP model used for receiving values.

Figure 3.3: Models used to measure throughput of ROS-LUNA connection implementations.

3.2.1 Native ROS port

The native ROS port provides access to ROS publishers and subscribers through the roscpp library and requires a ROS installation on the target system. This implementation only supports `bool`, `int`, `float`, `uint8`, `uint16` and `uint32` messages types. Reason for this is that the implementation assumes the existence of a single field named `data` in the ROS message. Since custom messages require the inclusion of a custom header with the message format, the choice was made to restrict the support for messages to TERRA's own data types. To use the native ROS port, the roscpp library must be manually added to the application's makefile.

To send and receive messages in a LUNA application, a send and polling loop are used. The rates of these loops are set to 100 Hz and can be adjusted in the source code generated by TERRA. Because of this polling rate an arbitrary delay of 0 ms to 10 ms is added to the latency of the transmitted data. Whether this is an issue or not depends on the application.

In Table 3.1, the amount of transmitted values per second is given for various desired send frequencies when using the native ROS port in LUNA to send or receive data. Although the frequency of the send and receive loops are 100 Hz, it is possible to send or receive values at a higher rate. These values are stored in a buffer and are send in groups. The rate measurements are done using the `rostopic hz` tool.

Table 3.1: Amount of transmitted values per second using native ROS ports in LUNA.

Sender frequency:	**100 Hz**	**250 Hz**	**500 Hz**	**1000 Hz**
Send in LUNA	99.9	250.4	500.2	1000.3
Receive in LUNA	100.0	249.5	499.9	1000.0

3.2.2 ROS-LUNA bridge

A background on the inner workings of the ROS-LUNA bridge is given in section 2.4.

Ridder (2018) states a few limitations in his thesis that were the reason to develop a native ROS port in LUNA. These reasons are: a fixed small buffer of 500 bytes that limits the amount of messages that can be send in a packet resulting in loss of messages when the buffer is not send before it overflows; the ROS node runs at a fixed polling frequency and a buffer issue on the LUNA side causing loss of messages when LUNA sends messages faster than the ROS side of the bridge can process.

The ROS side of the bridge suffers from issues that severely limit the throughput of the bridge. The throughput of the ROS-LUNA bridge is measured and the results of this are given in Table 3.2. From these measurements it is observed that the rate of sending values from LUNA to ROS is limited to 250 values per second at the default settings. With the loop rate of ROS side of the bridge being 750 Hz it is able to decode 250 values from LUNA to ROS per second. This suggest that only one value can be decoded per 3 iterations, which is confirmed by inspecting the code. For a few channels from LUNA to ROS this might not be a problem, but in larger systems with multiple channels this would require a undesirable high loop-rate. A change from decoding 1

value per 3 cycles to decoding all received values per cycle vastly improve the bridges throughput.

Table 3.2: Amount of transmitted values per second using ROS-LUNA bridge ports in LUNA.

Sender frequency:	**100 Hz**	**250 Hz**	**500 Hz**	**1000 Hz**
Send in LUNA	100.0	250.0	250.0	250.0
Receive in LUNA	100.0	250.0	499.6	998.9

Another limitation of the ROS-LUNA bridge is that only ROS 1 releases are supported. ROS Noetic Ninjemys, the latest and last distribution of ROS 1, is supported until May 2025, making this not an immediate issue. Due to the switch to DDS as middleware for ROS 2 and the dependence of the ROS-LUNA bridge on the ShapeShifter class from ROS `topic_tools` that allows for the run-time binding of topics, it is not trivial to port the bridge. For DDS these tools are not available.

3.2.3 Conclusion

The ROS-LUNA bridge is used during this thesis because it supports more complex message types and is usable on system that do not support ROS. The performance limitations mentioned in section 3.2.2 can and need to be solved.

3.3 Impact of imperfect networks

Wireless networks between multiple systems suffer from delay and packet loss. In this section the impact of these imperfections on latency of data transmitted between ROS and LUNA while using the ROS-LUNA bridge is analyzed.

To analyze the effects of network delay and packet loss on packets transmitted from ROS to LUNA, the latency between A and D in Figure 3.4 is measured. The setup to measure the latency is executed on a single system. Emulated delay and packet loss are added on the network between the ROS-side and the LUNA-side of the ROS-LUNA bridge by using `traffic control` in Linux. Detailed settings are given in Listing A.1 and A.2. The average latency between points A and D on the measurement setup without defining additional delay or packet loss is 0.25 ms as indicated by Figure A.2. All values are send at a fixed frequency of 100 Hz to represent a stream of set-points.

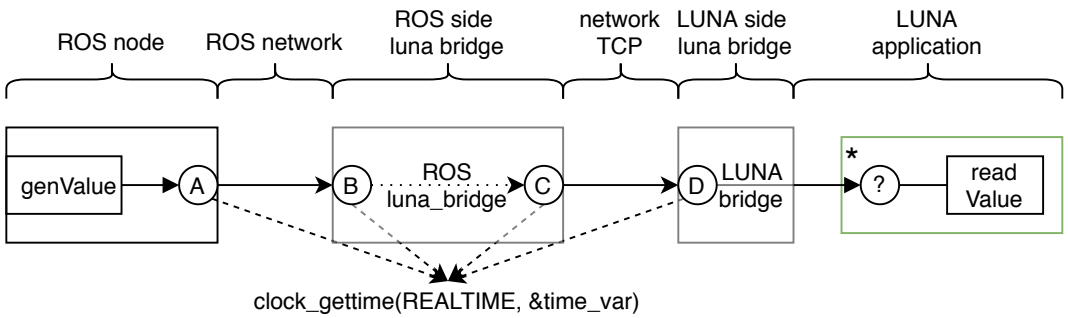


Figure 3.4: Diagram of the setup used for measuring latency between packets send from ROS to LUNA.

3.3.1 Packet loss

The measured latency of periodic packets between sending in ROS (A) and receiving in LUNA (D) with an emulated packet loss of 1% is given in Figure 3.5. It is observed that the latency spikes when a packet is lost and linearly reduces to the normal latency over a number of samples. This is due to the TCP protocol that guarantees orderly arrival of each packet. Successive

packets are delayed until the lost packet has arrived and as a result they arrive in a single burst on the receiving system.

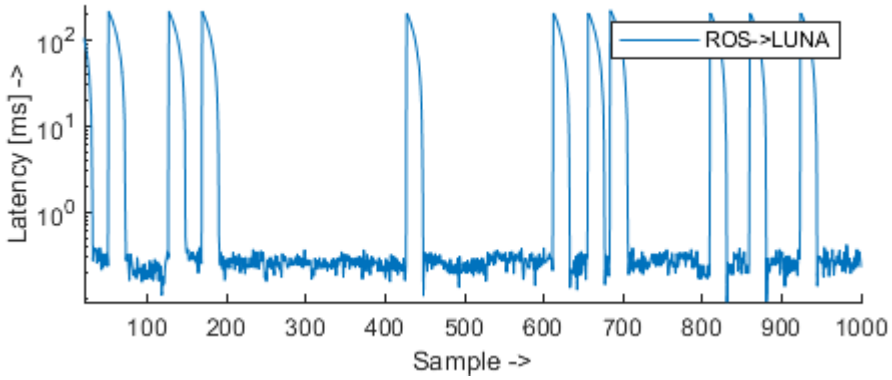


Figure 3.5: Measured effects of packet loss on latency between sending in ROS to receiving in LUNA.

The bursting due to packets that wait on arrival of previous packets is sketched in Figure 3.6. The circles on the *Send in ROS* and *Receive in LUNA* axes represents the send or receive time of the packets. The delay between sending and receiving packet 3 is highest, followed by 4 and 5 which each have T_{sample} less delay. Without re-transmission and by allowing the next packet to transmit at the original time, packet 4 would arrive significantly sooner than in this case.

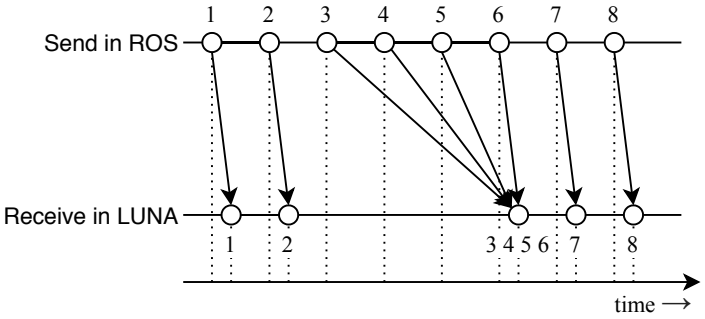


Figure 3.6: Sketched effects on latency from sending in ROS to receiving in LUNA when a TCP packet is lost and re-transmitted.

3.3.2 Delay

To analyze the real-world effects of Nagle's algorithm that is described in section 2.5 on periodic packets transmitted via the ROS-LUNA bridge, the latency is measured between sending in ROS (A) and receiving in LUNA (D). The latency is measured with an additional delay of 3 ms and 7.5 ms for 1000 samples. The results are given in Figure 3.7. It can be observed that the latency for the emulated delay of 3 ms, which is less than $\frac{1}{2}T_{transmit}$, is equal to the sum of the reference delay and the additional emulated delay. The latency for the emulated delay of 7.5 ms is between 12.5 ms and 27.5 ms, which is more than the sum of the reference delay and the additional emulated delay. In addition the measured delay shows a second saw-tooth pattern, which depends on the ratio between latency and transmission period.

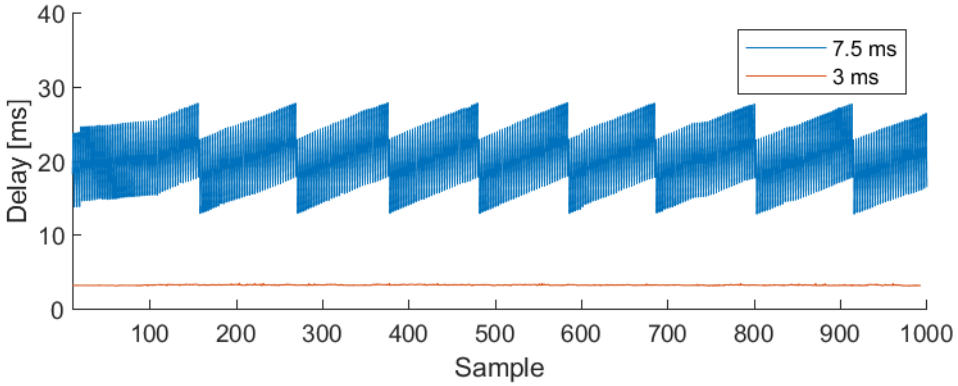


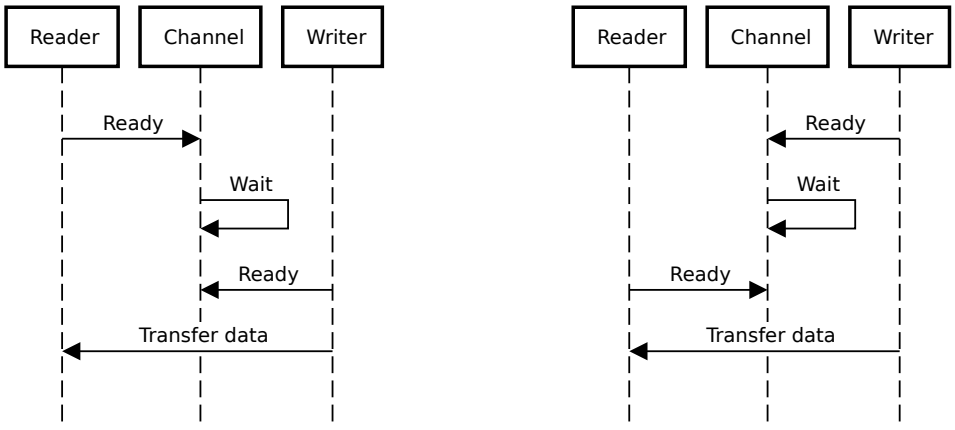
Figure 3.7: Measured effects of delay on latency of packets between sending in ROS and receiving in LUNA.

3.4 HRT-SRT connection

3.4.1 Difference between CSP and ROS channels

In LUNA, communication between processes is done using rendezvous channels. The channels implementing this behavior are called CSP channels in TERRA. Below properties and behavior of CSP channels depending on their configuration are given:

- **Blocking un-buffered CSP channels** use waiting-rendezvous communication to transfer data from a writer in one process to a reader in another process. For the transfer of data to occur, both writer and reader must be ready. An activity diagram for communication over a blocking unbuffered CSP channel is given in Figure 3.8.
- **Blocking buffered CSP channels** always accept new data from a writer. The oldest value in the buffer is overwritten when the buffer is full. A reader can read data from this channel as long as there is data in the buffer. When there is no data left in the buffer, the reader is blocked until the writer writes new data to the channel.
- **Non-blocking buffered CSP channels** always accept new data from a writer. The oldest value in the buffer is overwritten when the buffer is full. A reader can always read data from this channel, also when there are no new values in the buffer. In the latter case the most recent value written to the channel is read.



(a) Reader first. **(b)** Writer first.

Figure 3.8: Activity diagram of waiting-rendezvous communication.

The channels implementing communication via the ROS-LUNA bridge in TERRA are called ROS/LUNA bridge ports. Below properties and behavior of these ROS channels are given:

- **Incoming ROS channels** behave similar to buffered CSP channels and can be configured to be blocking or non-blocking. A blocking ROS channel makes the reader wait on new data when there is no data left in the buffer. A non-blocking ROS channel lets the reader read the latest value when there is no data left in the buffer.
- **Outgoing ROS channels** behave somewhat similar to buffered CSP channels, but have a shared buffer between all channels. With the channel property `send throug buffer` set to `false`, the data is transmitted during the write action of the writer. Otherwise, the data in the buffer must be send by explicitly calling the send function in a C++ code block. The user must take care that the function is called before the buffer is full.

3.4.2 Real-time requirements

When executing a LUNA application that uses the ROS-LUNA bridge on a system that uses the Xenomai co-kernel to meet real-time requirements, sending and receiving data at the LUNA side of the bridge causes unwanted mode switches from primary to secondary mode in real-time threads. In addition, new memory is allocated each time a packet is encoded. Sending data causes 2 mode switches per transmission whereas receiving data causes at least 4 mode switches per second with an additional 2 mode switches per received packet. These mode switches are cause by the TCP socket functions that are called for sending and receiving data.

One solution for mode switching from primary to secondary domain in a HRT process is to execute the TCP socket functions inside a lower priority real-time thread. Wijnholt (2017) and Ridder (2018) make use of separate threads for in- and outbound loops that are scheduled under a non real-time scheduling policy. Moving the network functions to a non real-time thread removes the mode switches, whereas moving them to a lower priority real-time thread removes the mode switches from within the HRT process.

To utilize the model-driven approach in TERRA, a separate lower priority process with secondary domain ROS-LUNA bridge code can be created for network functions. A general implementation of this idea is shown in Figure 3.9. This removes process where the send function with its mode switches to secondary domain is executed from the list of high priority threads. With this solution the mode switches to secondary domain caused by the socket listener remain. To prevent the execution of these switches in high priority threads, the socket listener must be executed in a lower priority thread. This solution must be implemented in the `ros-channels` component in LUNA.

Currently, it is possible to draw the pri-parallel construct in a TERRA model, but it is implemented in LUNA as a regular parallel construct. The code to assign priorities to CSP constructs is deactivated in LUNA during the port to Xenomai 3. To create and correctly execute the model in Figure 3.9 the assignment of priorities must be fixed.

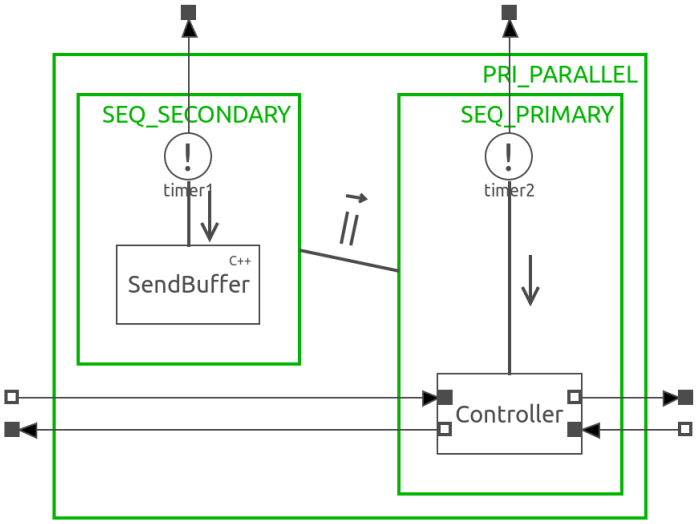


Figure 3.9: Model to execute network send functions in a lower priority thread.

Another option to incorporate a network connection that does not break the real-time requirements of the application is using RTnet. However, this only works with a limited set of Ethernet cards, which are not compatible with both the RaMstix and the used development machine. In addition, the focus of RTnet is on real-time networking whereas the network connection in this thesis does not have these requirements.

3.5 Conclusion and project requirements

The focus in this project is on further developing the ROS-LUNA bridge. The limited throughput from LUNA to ROS can be increased by decoding a complete packet upon arrival. The use of TCP as network protocol in the ROS-LUNA bridge reduces the utility of transmitted periodic packets over an unreliable network. Adding the option to use UDP as network protocol can be a solution for this.

Connecting the LUNA application to the bridge without considering the connection between SRT and HRT sub-systems can result in the HRT sub-system not meeting real-time requirements. The same goes for making Linux system calls from HRT code. SRT parts must be given a lower priority and connections between SRT and HRT components must be buffered and non-blocking. The selection of the network protocol must be done such that maximum utility is achieved for the signals.

3.5.1 Requirements

A set of requirements prioritized according to the MoSCoW method is given to achieve the goals stated in the introduction.

ROS-LUNA bridge

1. *The bridge must support UDP.*
   Delays and packet loss occur in a distributed CPS that uses a wireless network for communication. In some situations, receiving the newest data has priority over receiving all data. Using UDP as a transport protocol provides this feature since it does not check for arrival of the data.

2. *The bridge must provide an option to disable Nagle's algorithm.*
   Disabling Nagles algorithm when using TCP as transport protocol reduces bursting of

small packets. In periodic transmission of data, i.e. set points, simultaneous arriving of packets in a burst is undesired.

3. *An evaluation of the transport types must be done.*
   The effects of choosing different transport protocols in the ROS-LUNA bridge must be evaluated to validate the implementation and provide information for coupling between SRT tasks in ROS and HRT tasks in LUNA.

4. *The performance limitations must be solved.*
   The limitations found in section 3.2 must be overcome for the ROS-LUNA bridge to become usable.

5. *UDP and TCP should be usable in parallel.*
   In distributed CPS, communication channels between devices can have different requirements with respect to reliability and latency. This implies to provide using both UDP and TCP channels in a single application.

6. *A watchdog to detect network failure could be implemented.*
   When using UDP, both the ROS and LUNA side of the bridge have no information on whether the other side is still active or not. A watchdog in the bridge can be implemented to make the bridge aware of disconnection. An application-specific watchdog can also be implemented by users of TERRA.

7. *The bridge won't support ROS2.*
   Runtime binding of LUNA channels to ROS topics is done using the shape_shifter package that is available for ROS1. Since ROS2 uses DDS implementations from multiple vendors for communication between nodes such a package is not available.

8. *The ROS side of the bridge won't be code generated.*
   Code generation of the ROS side of the ROS-LUNA bridge as is done on the LUNA side of the bridge would reduce the dependence on ROS1. Also this would reduce the code complexity of the ROS side of the bridge and increase maintainability.

Connecting SRT and HRT tasks

1. *Generic SRT-HRT coupling method must be available.*
   A method for connecting SRT and HRT applications on multiple devices outside of CSP models in TERRA to achieve maximal utility is required for designing a complete application that uses the ROS-LUNA bridge.

2. *TERRA SRT-HRT coupling method must be available.*
   The ROS-LUNA bridge is intended to be used with HRT LUNA applications. For proper decoupling of the HRT processes from SRT processes, CSP models in TERRA can be constructed.

3. *The integration approaches must be verified.*
   To show that the real-time requirements of the HRT process are met when using the ROS bridge an evaluation must be done.

4. *HRT requirements of a model with SRT ports in TERRA should be verifiable.*
   Verification of a model with respect to real-time requirements is desired. Verification of CSP models with FDR only concludes that the model does not deadlock, but does not provide information on meeting timing requirements.

5. *A simulation approach that includes SRT components won't be done.*
   Simulation, either model/software/hardware-in-the-Loop, will not be done since the focus of this thesis is on the communication channel.

4 ROS-LUNA bridge refinements

In this chapter the improvements made to the ROS-LUNA bridge in the ROS node, LUNA library and TERRA tool to achieve the first project goal, *"Improve usability of the ROS-LUNA bridge in distributed Cyber-Physical Systems"* are discussed. Design and implementation is described first, followed by an evaluation of the new features.

4.1 Design and implementation

4.1.1 ROS side

The throughput of the ROS-LUNA bridge is increased by modifying the ROS side of the bridge. This is achieved by decoding a complete packet at the moment it arrives. This modification has increased the throughput of the bridge with a factor 3 when a single message is sent per packet. The throughput increases even further when multiple messages are combined in a single packet. Another modification that has been made is replacing the ROS spinOnce() function in the loop by a ROS asynchronous spinner in a separate thread.

An option to use UDP as transport protocol is added. In Figure 4.1 an UML diagrams representing the hierarchy of the TCP and UDP implementations are given.

When using TCP as transport layer, a single `LUNACommunicationTCP` object is instantiated when a connection request is received by the communication server. Upon connection a new TCP socket is created that is used by the `LUNACommunicationTCP` instance. Requests to connect ROS-channels in the LUNA application to a ROS topic are handled by this object and a new `TopicListener` or `RTBPublisher` object is created depending on the direction of the channel.

When using UDP as transport layer, there is no connection between the server and the client. The communication server for UDP in the ROS side of the bridge does not create a new socket for each LUNA application that tries to communicate with it. The client sends a connection packet to the server that instantiates a `LUNACommunicationUDP` object with the socket address and port of the client. When the request to connect a subscribing LUNA port to a topic is received by the server it calls the function of the `LUNACommunicationUDP` object that creates a new `TopicListener` object. When the request to connect a publishing LUNA port to a topic is received by the server it creates a new `RTBPublisher`.

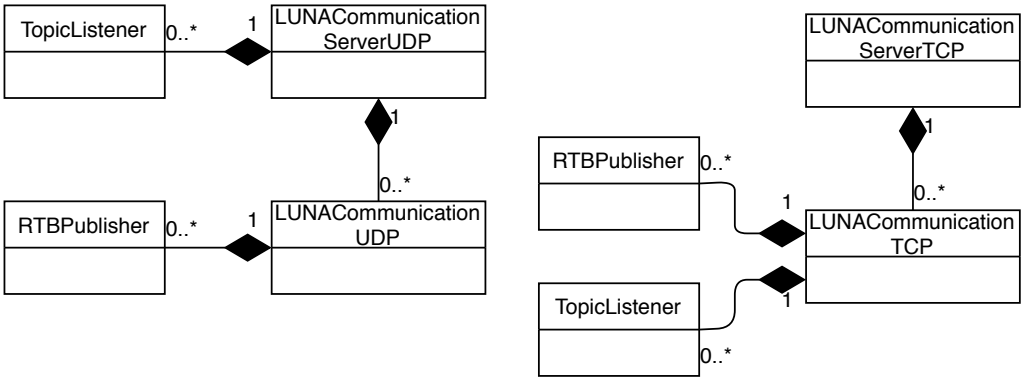


Figure 4.1: Differences in inner working of the bridge for TCP and UDP.

In ROS, a node can read parameters from a central parameter server that is accessible at runtime. Parameters can be configured in the command line when using `rosrun` or in the corresponding `.launch` file. The parameters that can be configured are: `use_tcp[bool]`,

`use_udp[bool]`, `disable_nagle[bool]` and `frequency[int]`. These parameters must be configured before launching the `luna_bridge`.

4.1.2 LUNA side

In LUNA, modifications are made to the `ros-channels` and `socket` components. The UDP receive function is updated such that it can be used non-blocking, similar to the TCP receive function in LUNA. In addition a function to send UDP packets to a specific address and a function to detect the address from which a message is received are added. A function to disable Nagle's algorithm has been added to the TCP class.

A `ROSChannelManagerUDP` class is added to the `ros-channels` component. Except for connection and closing the connection there are no differences in implementation between the TCP and UDP channel manager. The implementation of `ROSChannel` class is changed by replacing the hard coded TCP channel manager by a templated version. In the `connectToROS()` function an optional argument for disabling Nagles algorithm is added. This option affects all channels from LUNA to ROS that use TCP.

4.1.3 TERRA side

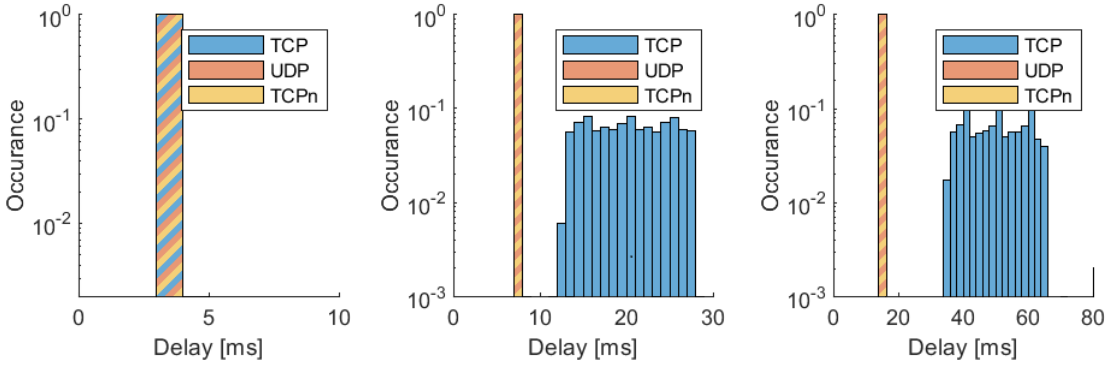
In TERRA, a transport type option for using UDP or TCP is added to the properties of a ROS-LUNA bridge port. This allows developers to select a transport type per port depending on which characteristics they require for each channel. TCP without Nagle's algorithm is not added as an option in TERRA and can only be added in code. This option would affect al TCP ports in the application and there currently is no option in TERRA to configure application-wide parameters.

4.2 Network effects on implemented transport types

The effect of network imperfections on communication via the ROS channels is different for the transport types. In this section the effects of delay and packet loss on channels with different transport types are evaluated. In the measurements, delay and packet loss are emulated with the use of traffic control in Linux. All these measurements are done using periodic packets with T_{sample} = 10 ms, representing a stream of set-points. More detailed measurements and documentation of the experiments are given in Appendix A.

4.2.1 Delay

In Figure 4.2, the measured latency between sending a message in ROS and receiving a message in LUNA is given for additional delays of 3 ms, 7.5 ms and 15 ms. For an additional delay of 3 ms, the measured latency for both TCP and UDP is equal to the sum of the reference and added delay. For an additional delay of 7.5 ms and 15 ms, the measured latency for UDP is also equal to the sum of the reference and added delay. When TCP is used, the measured latency is between 12.5 ms and 27.5 ms for an added delay of 7.5 ms, and between 35 ms and 65 ms for an added delay of 15 ms. The higher measured latency for TCP can be accounted to Nagle's algorithm that is discussed in section 2.5. Based on these measurements, the use of UDP or TCP with Nagle's algorithm disabled is the best option to be used in a distributed CPS that communicates periodic set-point data.



(a) Emulated delay of 3 ms between ROS and LUNA

(b) Emulated delay of 7.5 ms between ROS and LUNA

(c) Emulated delay of 15 ms between ROS and LUNAy

Figure 4.2: Measured latency between sending in ROS and receiving in LUNA via networks with different amount of delay. *TCP* represents the measurements that use TCP with Nagle's algorithm enabled whereas *TCPn* represents the measurements that use TCP with Nagle's algorithm disabled.

4.2.2 Packet loss

In Figure 4.3 the measured latency between sending a message in ROS and receiving a message in LUNA and the number of lost packets are given for an emulated packet loss of 1%. For UDP the latency and number of lost packets is as expected with a loss of 10 out of 1000 packets.

When TCP is used as the transport protocol, no packets are lost. However, around 25% of the packets is delayed. The reason for this is the ordered-arrival property of TCP packets. When a TCP packet does not arrive, a re-transmission is done after some timeout. After re-transmission, the set-points that are generated during this time are send to the receiver, which results in a burst of received packets. This behavior makes TCP unsuited for transmission of set-points over an imperfect network.

For TCP with Nagle's algorithm disabled 1% of the messages is lost. It is observed that the messages before the messages that did not arrive have a larger latency between 1 and 2 times the sample frequency. Sending set-points at lower frequencies confirmed this. When the time between samples is larger than the time between re-transmission no packets are lost.

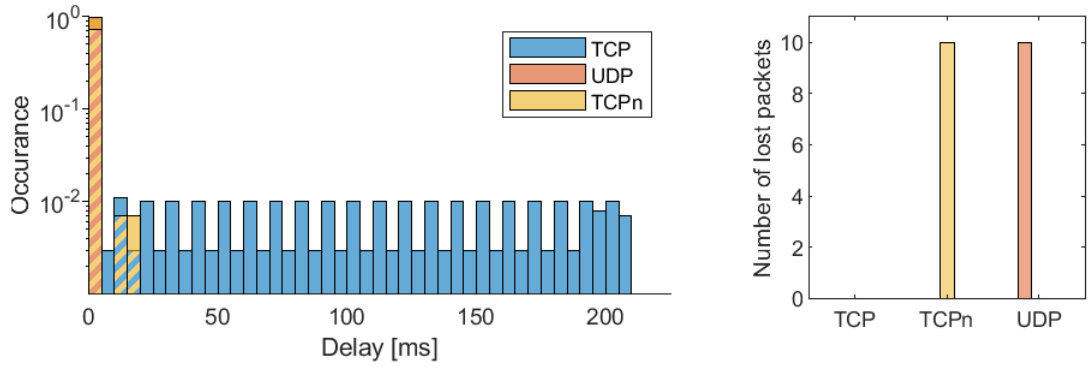
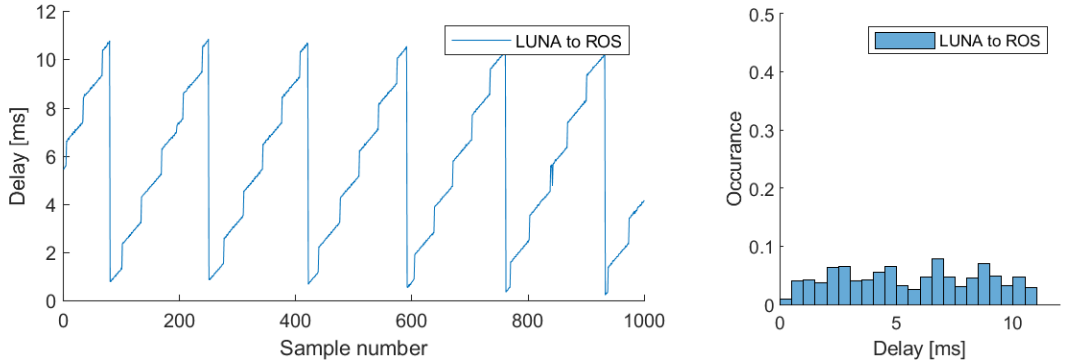


Figure 4.3: Measured latency between sending in ROS and receiving in LUNA via a network that has 1 % packet loss. *TCP* represents the measurements with Nagle's algorithm enabled whereas *TCPn* represents the measurements with Nagle's algorithm disabled.

4.3 Effects of SRT communication loop

As mentioned in section 3.4, one solution to prevent mode switching from primary to secondary domain caused by the ROS-LUNA bridge is to make use of separate threads for in- and outbound loops that are scheduled under a non real-time scheduling policy. This solution results in an additional delay, which is measured in this section.

In a LUNA application running on Xenomai that communicates with ROS using separate non real-time loops that sends and polls for messages at a fixed rate, mode switches no longer occur. As expected, this adds an additional delay with a that is bounded by $\frac{1}{f_{sendloop}}$ to the regular delay on the network channel. This can be observed in Figure 4.4, where the delay for messages sent via a 100 Hz publisher loop is given. In this measurement the values in the HRT process are also generated at 100 Hz. The saw-tooth that can be observed is likely caused by the use of different clocks in the LUNA application and ROS.



(a) Delay per sample.

(b) Distribution of delay over time.

Figure 4.4: Measured latency between sending in LUNA and receiving in ROS using a non real-time periodic sending loop.

4.4 Discussion

It can be observed in Section 4.2 that the addition of UDP and the option to disable Nagle's algorithm in TCP allow for a significant decrease of latency in imperfect networks with respect to the original TCP implementation. For applications where latency is of higher importance than reliability this addition leads to an increase of utility. The use of a non real-time communication loop results in an acceptable delay that is bounded by the period of the loop.

5 Connecting SRT and HRT tasks

In this chapter, methods to connect SRT and HRT tasks are given. The first section discusses generic methods to achieve the maximum utility of SRT computations that are sent via a network to a HRT system. This is followed by TERRA/LUNA-specific methods that connects SRT algorithms via communication channels with HRT control software using CSP models. In the third section these patterns are evaluated on a system with a real-time OS. In the last section the results are discussed.

5.1 Generic methods

5.1.1 Selection of network protocol

Within the scope of connecting SRT and HRT tasks on separate systems using a network there are two characteristics of the transmitted signals that are relevant in selecting the network protocol. These are sensitivity to latency and tolerance to packet loss. The requirements on these characteristics depend on the (sub-) system and signals that are transmitted over the link being classified as HRT, FRT, SRT or NRT. In Figure 5.1, the desired network protocol for different signal characteristics are given. Considerations for selecting a protocol for a given signal are given below.

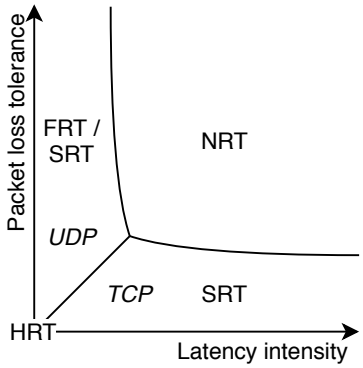


Figure 5.1: Desired network protocol and level of real-time for communication package characteristics.

Signals that are intolerant to latency and packet loss, need a network link that provides HRT guarantees. To achieve deterministic communication between multiple HRT systems dedicated hardware is required. UDP with several modifications can be used to achieve these characteristics (Kiszka et al., 2005). However, these signals are out of the scope of this thesis.

Signals that are intolerant to latency and somewhat tolerant to packet loss, need a network link with FRT or SRT specifications depending on the amount of tolerance. In the case of a system that publishes a series of set-points over an imperfect network with some packet loss the utility will be higher when using UDP. The ordered arrival of packets and re-transmission delay of TCP would decrease the utility of multiple set-points to zero and add latency spikes when packets are lost.

Signals that are somewhat tolerant to latency and intolerant to packet loss also require a network link with SRT specifications. One example of these signals is an instruction for a task sent from ROS to LUNA. The instruction needs to arrive on the other platform, but some latency is accepted. TCP guarantees that those packets arrive, but can add some latency when the packet is lost.

Signals that tolerant to latency and packet loss have no real-time requirements on the network link. A computed value keeps it utility and usually needs to be transmitted, but is allowed to be delayed.

5.1.2 Use of buffers

In Figure 5.2, the path of data between sending in a *Producer* application on one system and receiving in a *Consumer* application on another system is given. On several levels buffers are implemented. Some of these are in reach of the developer, where as others are defined in the kernel or hardware.

In user space the developer can configure in- or output buffers in the application. These buffers can be configured such that critical parts of real-time control applications can be executed without the need to directly interact with Linux system functions. By grouping data and locating the buffers at the right location, network use is minimized.

On the border of user and kernel space a socket buffer is located that allows user space applications to asynchronously write through a network socket. When the send queue is not full and data is written to the socket by the user application the system call will succeed. When the send queue is full and data is written to the socket by the user application the system call will block until there is space for the data. In kernel space TCP or UDP and IP information is added to the user data and sequentially stored in a transmit queue, called a queuing discipline or qdisc in Linux. Data from the qdisc(s) is mapped into the Tx ring buffer by the device driver of the Network Interface Controller (NIC). In the NIC, packets are stored in an internal buffer to match the packet rate with the physical rate of the network.

On the receiving side the packets are stored in the Rx ring buffer and read by the driver. The received data is processed on the IP and TCP/UDP layers and stored in the socket receiver buffer. In the user space consumer application a system call can be made to determine whether there is data is in the socket buffer. When the receive function is executed without data in the buffer it blocks until there is data in the socket buffer. When there is data in the socket buffer, the receive function will immediately return with the available data.

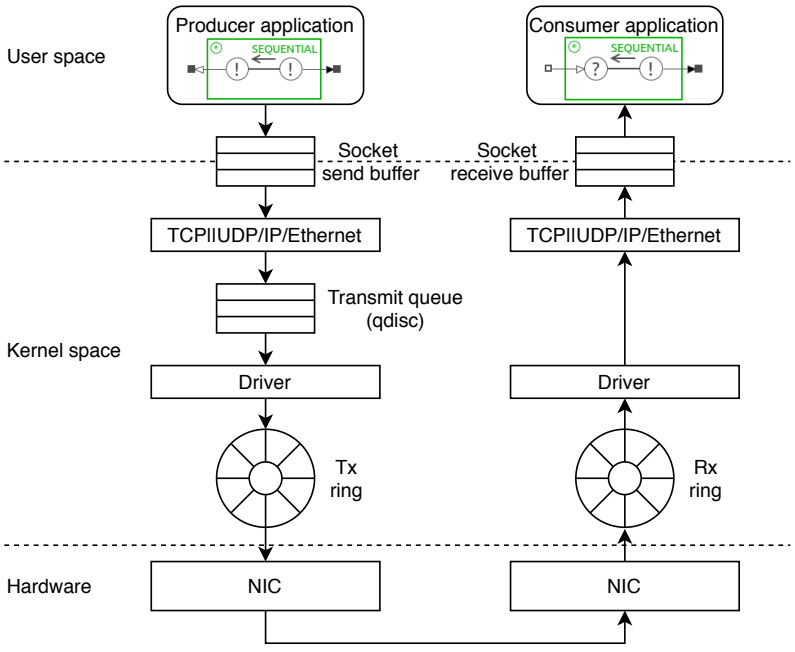


Figure 5.2: Transmission path between two application that are connected via a network.

5.2 TERRA/LUNA-specific methods

The TERRA-specific methods must provide a way to prioritize and decouple HRT process execution from SRT process execution. To achieve this there are two requirements: a reader or writer in a HRT process may never block and the treads used by a HRT process may never switch from primary to secondary mode in which the tread is executed by the regular Linux scheduler.

In this section, tool-specific methods are given to connect a HRT application in TERRA to a SRT application in ROS. This is done by introducing patterns for receiving and sending values from and to a ROS-channel. In Section 5.3 these patterns are evaluated on a real-time OS.

5.2.1 Receive patterns in TERRA

Receiving ROS channels in LUNA are either blocking or non-blocking and must always have a buffer with a size of at least one. Polling for messages is done in a separate thread, allowing the reader in a real-time process to read data without requiring a Linux system call. Blocking channels can be used for messages that only require processing once. For example, a sanity check can be performed on incoming messages. Since reading from a non-blocking ROS channel is a real-time safe operation, the benefits from performing this in a separate thread are lower resource usage and to keep the model organized.

In Figure 5.3, a CSP model is given in which the HRT loop controller directly reads from ROS channels. When non-blocking ROS channels are used, network calls do not interfere with execution the `SEQ_LOOP` HRT process. By using the default value of 1 as buffer size for both ROS channels the latest value is read by the readers. A larger buffer can be used, but has the consequence that latency between sending in ROS and receiving in the process in LUNA is increasing when the frequency of received values is higher than the frequency of the timer in `SEQ_LOOP`.

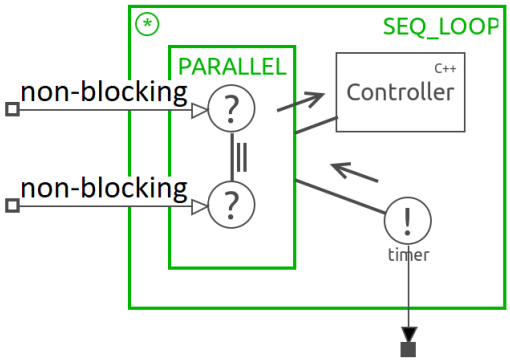


Figure 5.3: Model composition to receive values from ROS channel without blocking the hard real-time process.

Connecting a timed HRT process to a blocking (ROS) channel has the result that the HRT process misses deadlines since execution of the process is blocked until values from ROS are received. Using a separate lower priority process that receives and processes the received values is an option when the received values need additional processing in for instance a sequence controller. In Figure 5.4, a CSP model is given that implements that functionality. Timing of `SEQ_SEQEUNCE` is determined by the message rate of incoming messages from ROS. Timing of `SEQ_LOOP` is determined by the timer and will not be influenced by the message rate of incoming messages due to the use of non-blocking channels between SRT and HRT parts of the model.

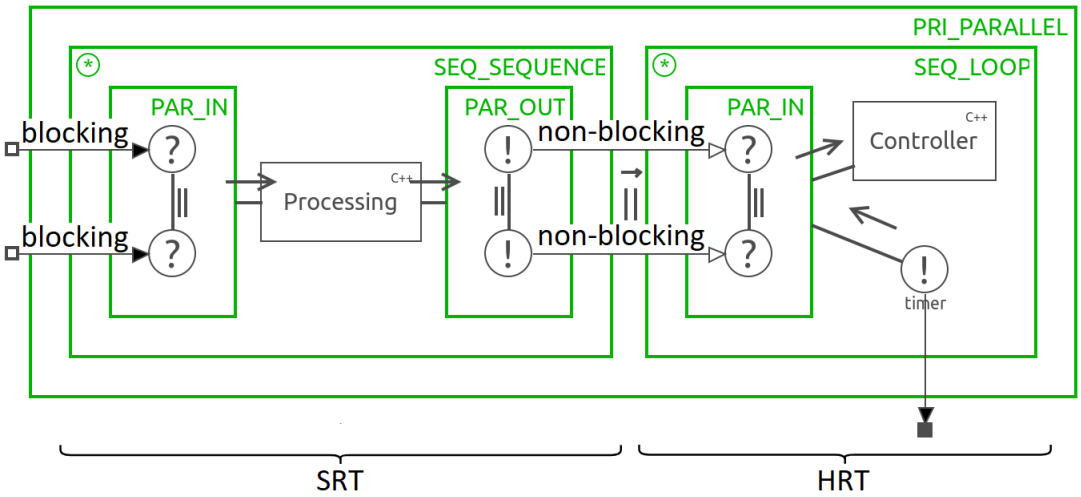


Figure 5.4: Model composition to receive values from a blocking channel in a lower priority process.

5.2.2 Send patterns in TERRA

Sending data from LUNA to ROS must not break the real-time requirements of the application. Therefore, the hard real-time process must be able to write values into a buffer without blocking or switching to secondary mode. To limit stressing the network, values can be grouped and send together or at a lower rate. CSP models for these two options are given and discussed below.

A naive way to interact between LUNA and ROS is to directly write to the outgoing ROS channel from a writer in a HRT process. Without using the separate non real-time send loop this would break the real-time guarantees of the HRT process.

All values

To send all values that are computed in the HRT process from LUNA to ROS, a separate lower-priority recursive process can be used. To ensure ordered transmission and that each value is sent exactly once, a blocked read from a buffered channel between the processes is done. This send pattern consists of sequential composition with `n` blocking readers in parallel, `n` writers in parallel and a C++ code block that sends the values in the ROSChannelManager's buffer to ROS. A CSP model of such a composition with `n = 2` is given in Figure 5.5.

In the `ROSChannelManager` there is a single buffer in which the values written to a ROS channel are stored until a call to the send function is made. In a large application there could be multiple processes from which data is written to this buffer. A call to send the data in the buffer to the ROS-side of the bridge must be made before the buffer is full. The call to send data from the buffer causes a switch from primary to secondary domain of the thread in which the operation is executed. Therefore, it is not possible to send the data in the buffer whenever a value is written to a full buffer, because this would cause the thread in which the writer that is writing to the channel to switch to secondary domain. This requires the developer to incorporate a codeblock that empties the buffer in the model.

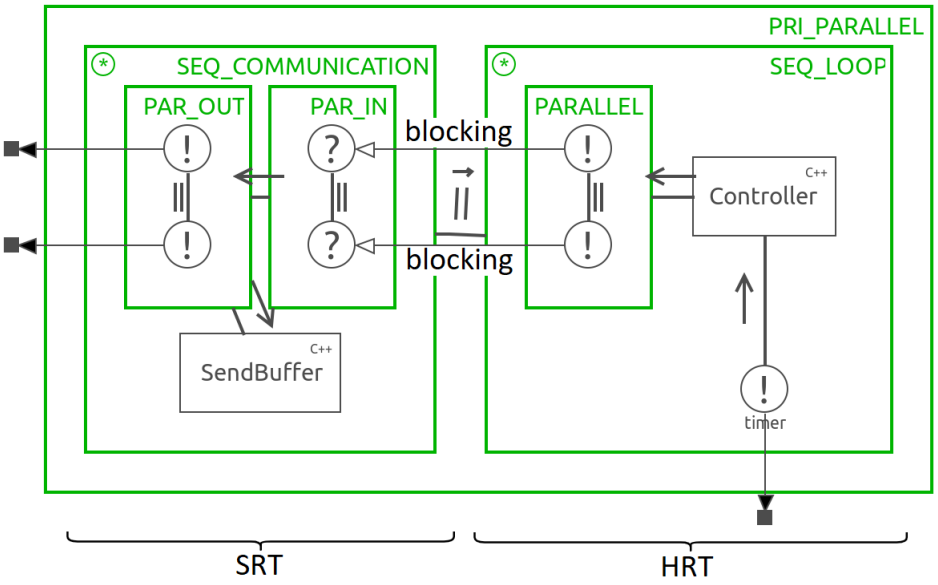


Figure 5.5: Model composition to send all values without affecting the hard real-time loops execution.

Sub-sampling

To send a fraction of all values that are computed in the hard real-time process from LUNA to ROS, a separate lower-priority timed recursive process can be used. Using a lower frequency timer, the latest value is read from a non-blocking buffered channel with a buffer size of 1 and sent to ROS. This send pattern consists of sequential composition with a writer connected to a timer port, `n` non-blocking readers in parallel, `n` writers in parallel and a C++ code block that sends the values in the ROSChannelManager's buffer to ROS. A CSP model of such a composition with `n = 2` is given in Figure 5.6.

The pattern to sub-sample values from the HRT process that sends the buffer of the ROSChannelManager in a separate timed process behaves similar to the experimental NRT communication loop discussed in section 4.1.2. The benefits of using a model-based approach are more control over which messages are combined and being able to control the send frequency.

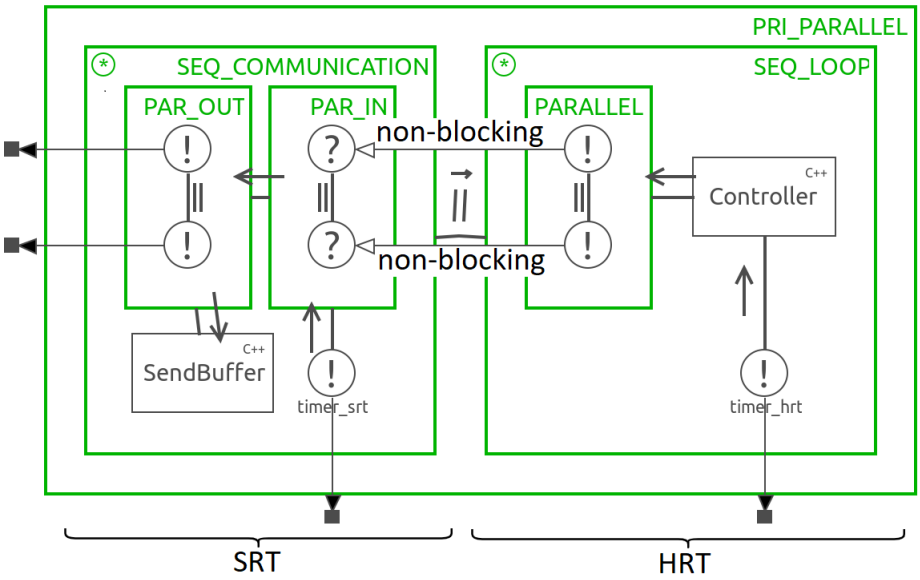


Figure 5.6: Model composition to send a fraction values without affecting the hard real-time loops execution.

5.3 Evaluation

The patterns to decouple the HRT process execution from SRT process execution are evaluated in this section. The measurements are done on a resource-rich platform with Ubuntu 18.04 and Xenomai 3.05 since the RaMstix embedded platform was not available during the evaluation of these patterns. The measurements are performed on a stressed system to achieve consistent results using `stress` with 8 workers spinning sqrt() (`--cpu 8`), 8 workes on sync() (`--io 8`), and 8 workers spinning on write()/unlink() (`--hdd 8`). This is executed using the following command:

```
stress -v --cpu 8 --io 8 --hdd 8
```

In Figure 5.7, the CSP model of the application that is used for reference measurements is given.

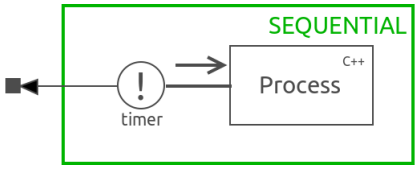


Figure 5.7: CSP model used for reference tests.

5.3.1 Receive patterns

In Figure 5.8 the jitter in the HRT process is given for several receiving patterns during 5000 cycles. In this figure, 'Reference' represents the measurements for the CSP model in Figure 5.7, 'Receive non-blocking' represents the measurements for the CSP model in Figure 5.3, 'Receive blocking' represents the measurements for the CSP model in Figure 5.4 and 'Receive naive' represents the measurements for a CSP model that does not implement any decoupling and reads directly from a blocking channel.

These results show that the jitter in the HRT process in naive implementation is significantly higher than the jitter in properly coupled implementations. This is expected because the HRT process is synchronized with the times that messages arrive from ROS. The maximum values are for Reference: 38.757 µs, for Receive non-blocking 39.841 µs, for Receive blocking: 45.480 µs and for Receive naive: up to one interval.

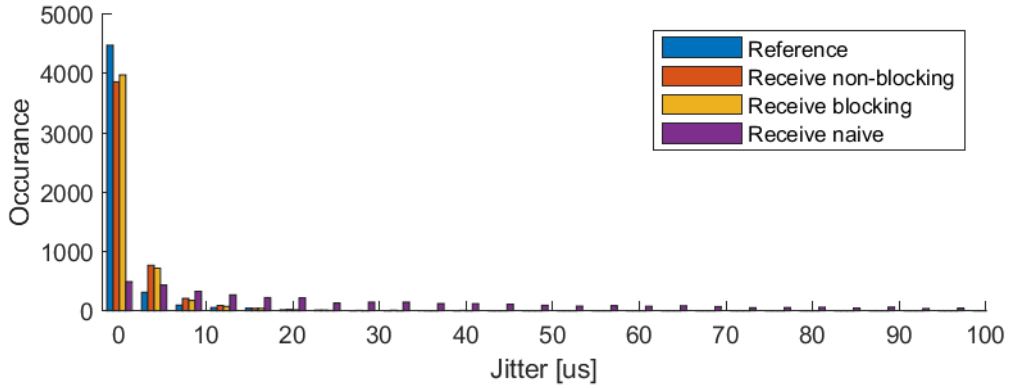


Figure 5.8: Absolute jitter for a HRT process with a cycle time of 1000 µs for several IO patterns used to receive data in LUNA to ROS.

5.3.2 Sending patterns

In this section, the sending patterns from section 5.2.2 are evaluated. In Figure 5.9 the jitter in HRT process is given for several sending patterns during 5000 cycles. In this figure, 'Refer-

ence' represents the measurements for the CSP model in Figure 5.7, 'Send naive' represents the measurements for the CSP model without any decoupling, 'Send buffered' represent the measurements for the CSP model in Figure 5.5 and 'Send sampled' represents the measurements for the CSP model in Figure 5.6.

These results do not show that the naive implementation has a significant effect on the jitter of the HRT process. The maximum values are for Reference: 38.757 µs, Send naive: 32.009 µs, Send buffered: 34.913 µs and Send sampled: 40.482 µs. With the Writer being last in the sequential `SEC_HRT` process and the execution of this process only taking a fraction of the interval between iterations these results can be expected.

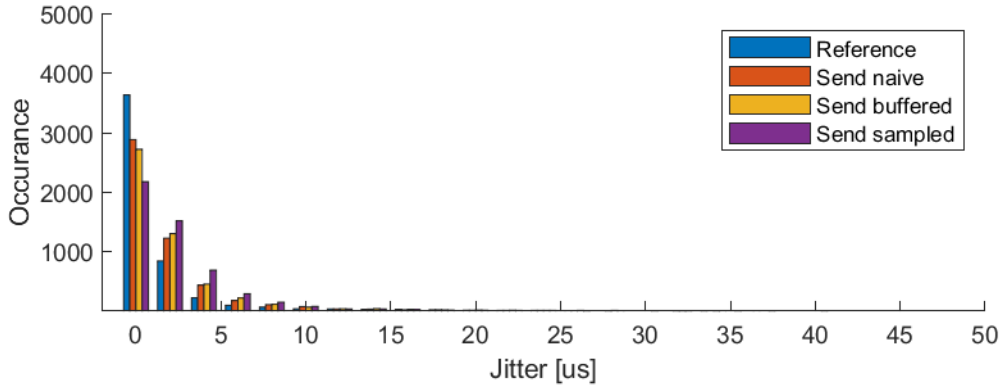


Figure 5.9: Absolute jitter for a HRT process with a cycle time of 1000 µs for several IO patterns used to send data from LUNA to ROS.

In Figure 5.10, the jitter in the HRT process is given for an application that uses a SRT send loop inside the `ROSChannelManager`. This send loop calls the send function of the `ROSChannelManager` at a predetermined rate. 'Reference' represents the measurements for the CSP model in Figure 5.7, 'Send 100 Hz' represents the measurements with a SRT send loop at 100 Hz and 'Send 1000 Hz' represents the measurements with a SRT send loop at 1000 Hz. The maximum measured values are for Reference: 38.757 µs, for Send 100 Hz: 26.9870 µs and for Send 1000 Hz: 31.899 µs.

The observed jitter for using a SRT send loop in `ROSChannelManager` is lower than the jitter observed in the applications that use the send patterns given in section 5.2.2 and similar to the jitter of the reference HRT process. It is expected that the model-based approach has a lower code efficiency that causes a higher jitter for the HRT process when using send patterns.

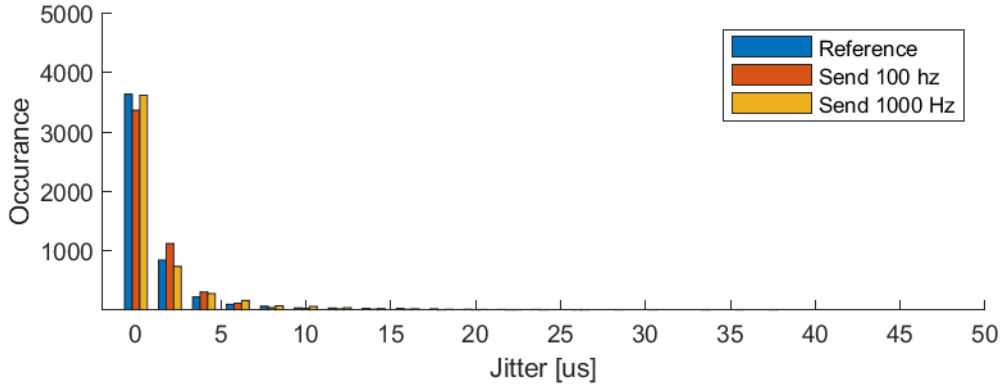


Figure 5.10: Absolute jitter for a HRT process with a cycle time of 1000 µs that sends data from LUNA to ROS by using for code based decoupling in LUNA.

5.4 Discussion

As expected, receiving from ROS in a naive way causes significant jitter to the HRT process due to the synchronization between the execution of the HRT process and the time of arriving messages. When using the blocking or non-blocking receive pattern, timing of the HRT process is not disrupted.

Sending messages to ROS from a HRT loop in a naive does not cause significant jitter, which can be attributed to the abundance of resources available on the resource-rich platform. When using the buffered or sample send pattern, the timing of the HRT process is not disrupted. The use of a SRT send loop in code results in less jitter than when a model-based SRT send loop is used. A cause for this can be the overhead that is introduced by using the model-based approach.

The difference in results when using proper coupling patterns or naive connections is insignificant, which can be explained by the abundance of computing resources on the device used for measurements. Similar experiments should be done on a RaMstix embedded platform to evaluate the impact of properly or naively coupling SRT and HRT processes on a device with limited resources.

6 Conclusion and Recommendations

Conclusion

The goal of this project was to increase the usability of the ROS-LUNA bridge and provide means to integrate soft real-time and hard real-time software in the design process of Cyber-Physical Systems.

Usability of the brigde is improved by adding support for UDP and an option to disable Nagle's algorithm in TCP. The added value of the UDP option in the ROS-LUNA bridge is shown in a comparative evaluation of both transport types. In addition, severe performance limitations of the bridge are overcome by decreasing the number of cycles needed to decode a network packet in the ROS side of the bridge.

IO patterns to communicate via communication channels with soft real-time software are developed to aid in integration of hard and soft real-time tasks in Cyber-Physical Systems. It is shown that by using these IO patterns, real-time requirements of critical tasks are met. By using a model-driven approach in connecting the communication channels the developer remains in control over the way the HRT processes interact with the SRT processes.

Recommendations

Due to the covid-19 restrictions, an implementation of embedded control software using the improved ROS-LUNA bridge for a representative demonstration setup could not be achieved. The use of this work in such an use case is expected to produce valuable feedback that can be used in maturing the bridge.

The thread implementation in LUNA could be revisited to provide full support for non real-time and real-time threads in Xenomai 3. Currently, threads are created using POSIX functions, but thread priorities of Xenomai are not correctly implemented. Also, considering to limit the support of LUNA for only Xenomai 3 as a real-time OS and deprecating code for other OS's that are no longer used at RaM is expected to significantly increase maintainability of the LUNA library.

Within TERRA there is no visual difference between blocking and non-blocking channels. This critical property of channels is also not shared between multiple sub-models. Also, blocking and buffer properties on in- and outgoing channels cannot be configured from within sub-models, creating room for errors.

On the road to a tool coverage for all layers of embedded control software the switch to a code generated interface on both ROS and LUNA side is recommended. Recently, Eclipse plugins for model-driven design with ROS 2 have emerged. This would mean that only a network interface in LUNA is required that could be connected to a network interface in other tools.

A Bridge measurements

In this appendix, detailed information on the latency measurements that are performed on the bridge with various emulated delays and loss on the network is given. First, the measurement setup is given. This is followed by the raw measurements for the graphs in Chapter 5.

The used measurement setup is given in Figure A.1. In here eight points, A to H, are indicated to specify the locations where timestamps for each sample are taken. This setup allows for measuring the round trip time (RTT) and the one way delay (OWD) in both directions. For RTT, the time between A and H is meant and for OWD the time between A and D or E and H, called OWD_{AD} or OWD_{EH} from now on.

The measurements in this appendix are all done on a periodic stream of values with a frequency of 100 Hz.

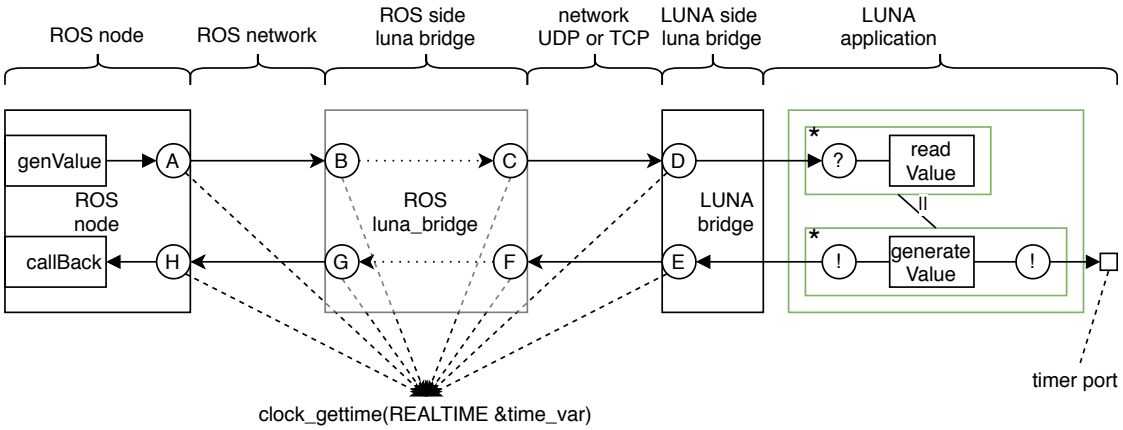


Figure A.1: Measurement points inside the ROS-LUNA bridge.

To represent a real-life scenario where the LUNA application is executed on the embedded device and the ROS application and ROS-LUNA bridge on the resource rich device, network emulation must only be done on the link between C and D, and E and F. For emulation of delay and packet loss on only a specific port number, `traffic control` is used with the settings given in Listing A.1 and A.2.

```
tc qdisc add dev lo root handle 1: prio priomap 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0
tc qdisc add dev lo parent 1:2 handle 20: netem delay $DELAY ms
tc filter add dev lo parent 1:0 protocol ip u32 match ip sport
    $PT_NUMBER 0xffff flowid 1:2
```

Listing A.1: Commands used to emulate delay on network port

```
tc qdisc add dev lo root handle 1: prio priomap 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0
tc qdisc add dev lo parent 1:2 handle 20: netem loss $LOSS %
tc filter add dev lo parent 1:0 protocol ip u32 match ip sport
    $PT_NUMBER 0xffff flowid 1:2
```

Listing A.2: Commands used to emulate packet loss on network port

Reference measurements

In Figures A.2 and A.3, the latency for all points the system is given with respect to measurement point A. In this figure can be seen that the time it takes to send a value from ROS to the LUNA application and back takes less than 1 ms under ideal circumstances.

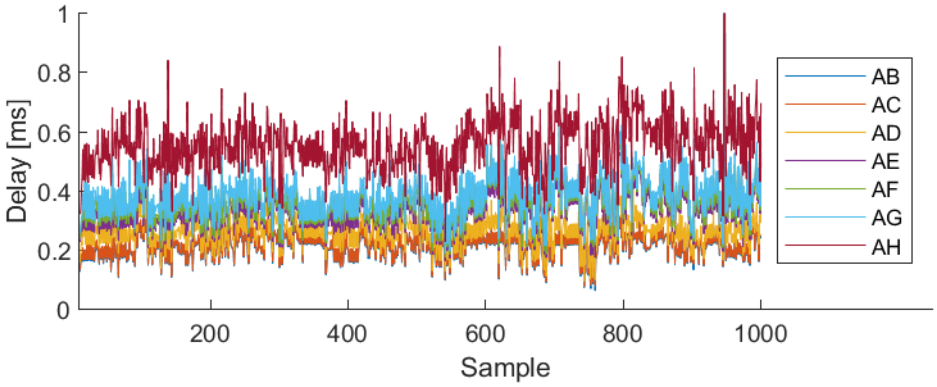


Figure A.2: Reference measurement for TCP.

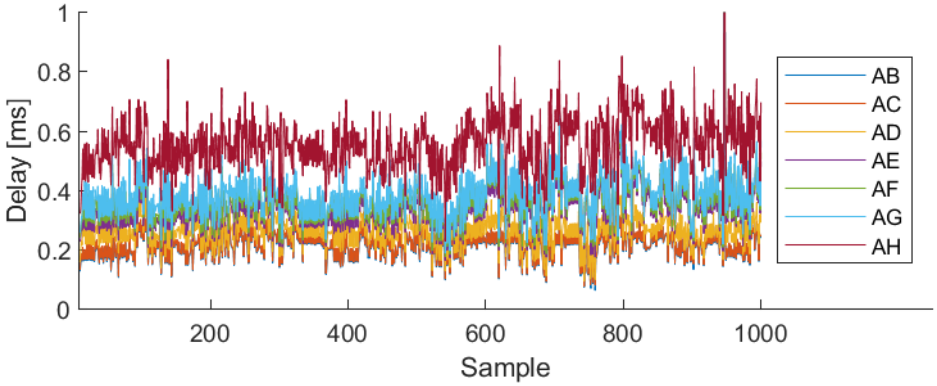


Figure A.3: Reference measurement for UDP.

Measurements with emulated delay

In Figure A.4, the measured OWD_{AD} with an additional emulated delay of 3 ms is given for 1000 samples. With UDP as transport type, the latency is between 3 ms and 4 ms. With TCP as transport type and Nagle's algorithm disabled, the latency is between 3 ms and 4 ms. With TCP as transport type and Nagle's algorithm enabled, the latency is between 3 ms and 4 ms.

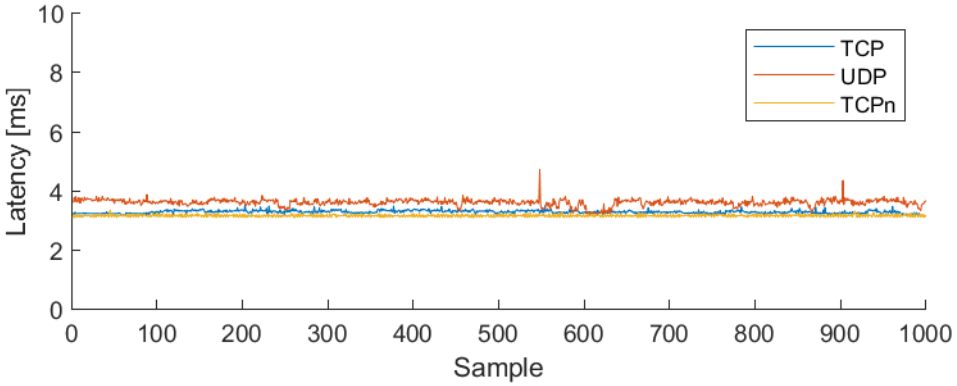


Figure A.4: Latency for an emulated delay of 3 ms. *TCP* represents the measurements with Nagle's algorithm enabled whereas *TCPn* represents the measurements with Nagle's algorithm disabled.

In Figure A.5, the measured OWD_{AD} with an additional emulated delay of 7.5 ms is given for 1000 samples. With UDP as transport type, the latency is between 7.5 ms and 8.5 ms. With TCP as transport type and Nagle's algorithm disabled, the latency is between 7.5 ms and 8.5 ms. With TCP as transport type and Nagle's algorithm enabled, the latency is between 12.5 ms and 27.5 ms.

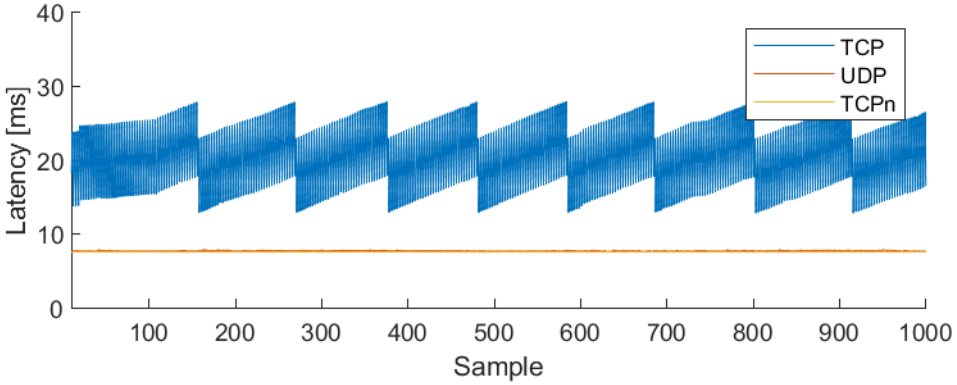


Figure A.5: Latency for an emulated delay of 7.5 ms. *TCP* represents the measurements with Nagle's algorithm enabled whereas *TCPn* represents the measurements with Nagle's algorithm disabled.

In Figure A.4, the measured OWD_{AD} with an additional emulated delay of 15 ms is given for 1000 samples. With UDP as transport type, the latency is between 15 ms and 16 ms. With TCP as transport type and Nagle's algorithm disabled, the latency is also between 15 ms and 16 ms. With TCP as transport type and Nagle's algorithm enable, the latency is between 35 ms and 65 ms.

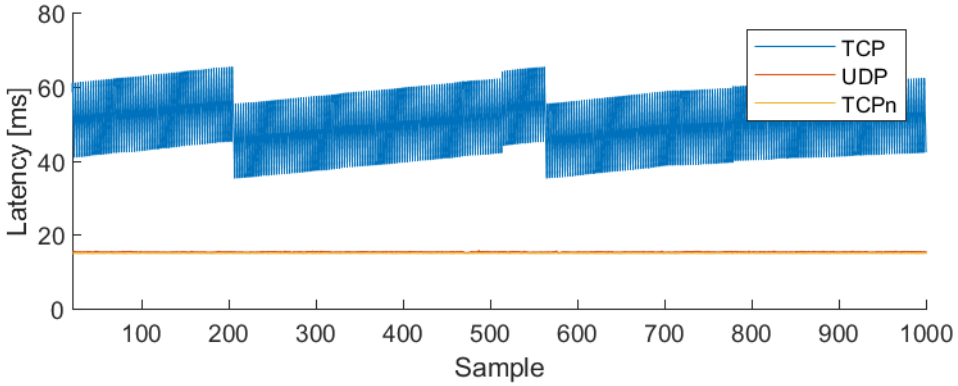


Figure A.6: Latency for an emulated delay of 15 ms. *TCP* represents the measurements with Nagle's algorithm enabled whereas *TCPn* represents the measurements with Nagle's algorithm disabled.

Measurements with emulated packet loss

In Figure A.4, the measured OWD_{AD} with an additional emulated loss of 1% is given for 1000 samples. With UDP as transport type, the loss of a packet only means the loss of a sample. With TCP as transport type and Nagle's algorithm disabled, the loss of a packet only means the loss of a sample. With TCP as transport type and Nagle's algorithm enabled, the loss of a packet results in a re-transmission after some timeout. This timeout and the ordered delivery property of TCP cause that the sample in the lost packet and all samples between the moment that the lost sample is sent and resend are delayed. In the measurements this can be observed by the delay peaks with a width of around 20 samples.

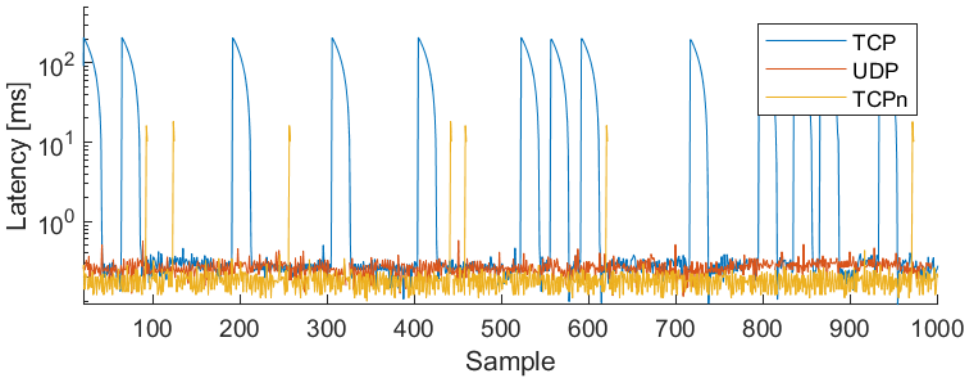


Figure A.7: Latency for an emulated packet loss of 1%. *TCP* represents the measurements with Nagle's algorithm enabled whereas *TCPn* represents the measurements with Nagle's algorithm disabled.

B Practical notes on using ROS-LUNA bridge

B.1 ROS

The bridge can be used with most of the recent ROS (1) distributions, but during this thesis ROS Melodic Morenia is used on Ubuntu 18.04.

B.1.1 Installation

1. Download and copy the `luna_bridge` and `ros-raw_message` packets to the src folder of a catkin workspace.
2. Define the location of the LUNA executable and header folder before compiling the catkin workspace by using the following commands:

```
export LUNA_LIBRARY_DIR = /<folder with libLUNA.a>
export LUNA_HEADERS_DIR = /<folder with luna-config.h>
```

3. Source the ROS environment:

```
source /<catkin workspace folder>
```

4. Compile the catkin workspace using:

```
catkin_make
```

B.1.2 First start

1. Run the following command in a sourced terminal:

```
ros_launch luna_bridge luna_bridge.launch
```

B.1.3 Known issues

- When the ROS source of the ROS-LUNA bridge is not directly downloaded from git into the catkin workspace it can occur that some files have incorrect permissions.
  - This can be solve by executing the following command in the catkin_ws/src folder:

```
sudo chmod -R 751 ros-luna-bridge
```

B.2 TERRA

In Figure B.1 the properties of a ROS-LUNA bridge channel in TERRA are given.

- **Buffer size:** Modifying this property only affects receiving channels in TERRA. The buffer size must be at least 1.
- **Field name:** The name of the field of a message in which data is stored.
- **Is blocking:** Modifying this property only affects receiving channels in TERRA. Determines whether a channel blocks until a value is received or can be read from at all times.
- **Msg type:** The type of the message in which data is stored.

- **Send through buffer:** Modifying this property only affects sending channels in TERRA. If this property is set to `true`, a explicit call to send the data must be made in a C++ code block.
- **Topic name:** Name of the topic to which the channel publishes or subscribes.
- **Transport type:** Transport type that is used to transmit data over the channel. Must be `TCP` or `UDP`.

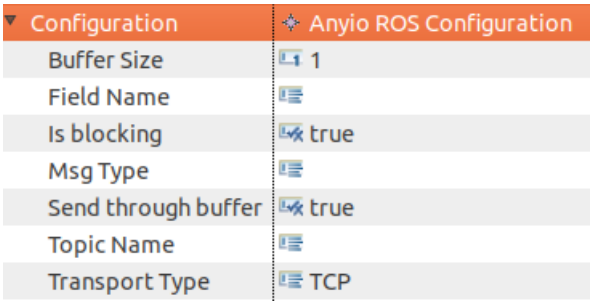


Figure B.1: Properties of the ROS-LUNA bridge channels in TERRA.

To use the ROS channels with TCP and Nagle's algorithm disabled, the field `no_delay` in the function below must be set to `true`. This function can be found in the file: `MainModel.cpp`. By default, the value for `no_delay` is `false`.

```
int ROSChannelManagerTCP::connectToROS(const char *hostname,
                                       const int port,
                                       const bool nodelay)
```

B.2.1 Adding priorities

Priorities can be added to the CSP construct by adding the following lines of code in the protected region for manual tree modification of `main.cpp`:

```
term->setPriority(<desired priority>);
model->setPriority(<desired priority>);
```

All CSP constructs in the tree will now get a their priority assigned.

Bibliography

Bezemer, M. (2013), *Cyber-physical systems software development: way of working and tool suite*, Ph.D. thesis, University of Twente, Netherlands, doi:10.3990/1.9789036518796.

Boode, A. (2018), On the automation of periodic hard real-time processes: a graph-theoretical approach, University of Twente, Netherlands, number 18-466 in IDS Ph.D-Thesis Series, ISBN 978-90-365-4551-8, doi:10.3990/1.9789036545518.

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*, IEEE, pp. 73–77.

Hoare, C. A. R. (1978), Communicating sequential processes, **vol. 21**, no.8, pp. 666–677.

Kempenaar, J. (2014), *Communication Component for Multiplatform Distribution of Control Algorithms*, Master's thesis, University of Twente.

Kiszka, J., B. Wagner, Y. Zhang and J. Broenink (2005), RTnet – A Flexible Hard Real-Time Networking Framework, doi:10.1109/ETFA.2005.1612559.

Ridder, L. (2018), *Improvements to a tool-chain for model-driven design of Embedded Control Software*, Master's thesis, University of Twente.

Vos, P. (2015), *Demonstrator combining ROS/TERRA-LUNA*, Master's thesis, University of Twente.

Werff, W. (2016), *Connecting ROS to the LUNA embedded real-time framework*, Master's thesis, University of Twente.

Wijnholt, R. (2017), *Design of a real-time network channel in LUNA*, Master's thesis, University of Twente.

Xenomai (2020), Introduction Start_Here, https://gitlab.denx.de/Xenomai/xenomai/-/wikis/Start_Here, accessed: 2020-08-28.