Dynamic Offloading of Control Algorithms to the Edge using 5G and WebAssembly

Ahmed Dhiaa Tariq Al Bayati

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6231
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2024 Ahmed Dhiaa Tariq Al Bayati. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2024

To God
& my parents

Abstract

The aim of this work was to test if WebAssembly universal byte code and 5G communication technology is suitable in the context of offloading control mission of real-time systems. To test these tools a new dynamic offloading framework was implemented and tested on a Furuta pendulum, an inherently unstable and time-critical process using, among other computing units, an edge node as the offloading target. The implementation is considered dynamic because: 1) The local device which interacts with the I/O of the process dynamically send the control application to be used by the edge node. 2) The local device dynamically decides on which controller should control the process, either the local fallback linear quadratic regulator controller or the CVXGEN Model Predictive Control solver written in an Ahead-of-Time WebAssembly format, which is used by the edge node. The work concluded that Wasm run in interpreted form was too slow to control the process but AOT compiled Wasm which could be run by WasmEdge runtime worked well with an execution time close to the native speed. It was also concluded that data transfer using 5G technology without uRLCC was fast enough to balance the pendulum and is suitable for offloading, other communication medium where also tested in this work such as: Wi-Fi and cabled Ethernet. In the study we also developed a control quality measure for decision making on when to offload.

Acknowledgements

I wish to extend my sincere appreciation to my supervisor Karl-Erik Årzén, whose support has granted me an extraordinary opportunity of. Karl-Erik consistently allocates time for updates on my ongoing work and readily offers assistance during challenging moments.

Special thanks goes to my co-supervisor and examiner Johan Eker for his continuous support, guidance, and kindness.

My heartfelt gratitude extends to my family, whose enduring support and genuine interest in my work have been constant since day one. Without the love, support, and prayers from Dhiaa Al Bayati (my dad), Sundes Al Bayati (my mom), Zahra Al Bayati (my older sister), Mohammad Al Bayati (my younger brother), Murtadha Al Bayati (my younger brother), and Fatima Bahi (my sister's daughter), I wouldn't be where I am today.

I would also like to express my thanks to the numerous individuals in the Department of Automatic Control at LTH, Ericsson, and the Department of Electrical and Information Technology at LTH for their invaluable assistance. Special acknowledgments go to Felix Agner, Anders Blomdell, Bo Bernhardsson, Fethi Bencherki, Max Nyberg Carlsson, Marko Guberina, Albin Heimerson, Raihan Ul Islam, Olle Kjellqvist, Yiannis Karayiannidis, Anders Nilsson, Haorui Peng, Richard Pates, Marlén Robbani, Emil Sundström, and all others who supported and wished the best for me.

Financial Support

This work has received funding from Vinnova through the AORTA project. The author is WASP affiliated and also a member of ELLIIT.

Contents

1. Introduction **11**
1.1 Research Questions 12
1.2 Contributions 12
1.3 Context of the Thesis 12
1.4 Thesis Outline 13
1.5 Related Work 13

2. Controller Offloading **16**

3. Background **20**
3.1 Furuta Pendulum 20
3.2 Model Predictive Control 21
3.3 Linear Quadratic Regulator 23
3.4 WebAssembly 24
3.5 Containers 25
3.6 5G 25
3.7 Edge Computing 26

4. Implementation **27**
4.1 Experimental Setup 27
4.2 Conceptual overview 28
4.3 Model 30
4.4 CVXGEN 33
4.5 Control Quality 34
4.6 Local device (Host) 36
4.7 Remote Solver 41
4.8 Software overview 43

5. Results **44**
5.1 Timing for the Control Signal 44
5.2 Timing of the File Sending 54
5.3 Control Quality 56
5.4 Demonstration 57

6. Thesis Summary **58**
6.1 Weaknesses and Potential Criticism 58
6.2 Future Work & Improvements 59

Bibliography **62**

A. Implementation **66**
A.1 Host 66
A.2 Remote Solver 90
A.3 AOT Wasm MPC Solver 109

1

Introduction

Most controllers are currently executed physically close to the process they are controlling. There are several reasons for this. These include the need for a short round-trip time (RRT), and the simplicity of the implementation.

While these reasons hold merit, a significant constraint of locally implemented controllers arises from the limitations of the device hosting the controller, e.g., computation and storage constraints. These limitations may not be a significant issue for processes that do not require an advanced controller, e.g. low-order processes without challenging specifications. However, it can become problematic when dealing with processes requiring an advanced controller that requires substantial computations and/or storage.

To address these limitations, the cloud computing paradigm is a promising candidate solution for compute or storage intensive applications, e.g., optimization-based controllers or learning-based computer vision. Although cloud computing offers enhanced storage and computing capacity, it also introduces potential challenges such as high and unpredictable latencies. In contrast, edge computing, a paradigm situating the execution environment closer to the data source than traditional cloud computing, presents a compelling alternative [Årzén et al., 2018].

The offloading strategy presented in this thesis has the potential to not only increase the performance, but also reduce power consumption and extending the capabilities of the controller by enabling the execution of advanced services traditionally confined to the cloud in close proximity to the controller. An additional advantage lies in the potential cost reduction for local devices, which can be equipped with more economical, less powerful hardware which can utilise the edge for control or other services that require additional resources.

1.1 Research Questions

This work aims to explore dynamically offloading parts of a control algorithm compiled to Ahead-of-Time WebAssembly to a remote computing unit, usually called the remote solver, and testing different communication mediums such as 5G, WiFi, and cabled-Ethernet by controlling a Furuta pendulum in the upward position. More concretely, this thesis aims to address the following research questions:

- How would a dynamic offloading strategy utilizing WebAssembly look like?
- What are the expected delays and control qualities of the different communication mediums and computing resources?
- How to evaluate different controllers during runtime?

1.2 Contributions

The outcome of this thesis could be summarised as follows:

- **Offloading Strategy:** The thesis presents how a dynamic offloading can be implemented by dividing the Offloading Strategy into different logical units which could be implemented independently.
- **Demonstration:** The thesis showcases the feasibility and a proof of the offloading strategy by implementing it and testing it on a real system.
- **Control quality measure:** The thesis proposes a simple control quality measure that can be used to make an informed decision on when to offload and onload, i.e., to go back to local execution.

1.3 Context of the Thesis

This work is part of a larger project called Advanced Offloading for Real-time Applications (AORTA), financed by Vinnova. The aim of the project is to develop and demonstrate the feasibility and advantages of 5G for offloading to the edge. The partners in the AORTA project are Cognibotics AB, Ericsson AB, Lund University, and Mälardalen University working together to showcase the potential of offloading.

1.4 Thesis Outline

Chapter 1 introduces the thesis, Chapter 2 motivates the field and this work, Chapter 3 present an overview of relevant topics used in this work,Chapter 4 presents the offloading implementation,in Chapter 5 the obtained results are presented and discussed and in Chapter 6 the thesis is concluded.

1.5 Related Work

Some of the related work in the area of offloading and cloud computing are presented in this section. The related work could be divided crudely into different sub-sections. Some works, such as the doctoral thesis of [Skarin, 2021] and [Peng, 2023], cover multiple sub-section but for simplicity will be only included in one of the sub-sections. The different areas that will be presented in this sections are: Overview, Offloading, Offloading decision, Infrastructure and WebAssembly.

Overview

In this sub-sections the works that have a more of an overview of the field and/or investigate multiple areas of the offloading problem are presented.

One notable contribution to this field is the work by Per Skarin [Skarin, 2021]. In his study, Skarin investigated various aspects of offloading and cloud computing. One of the testbeds used involved controlling a ball and beam process using 5G and an edge node. Haorui Peng's work [Peng, 2023] is also closely related to the thesis. She explores the integration of cloud infrastructure into traditional systems and investigates the possibilities and limitations of cloud integration. Kumar et al., [Kumar et al., 2012], did a survey exploring computational offloading for mobile systems between 1996-2012. The authors say that the research was focusing on feasibility between 1996-2000, while the focuses was more on offloading decision and the infrastructure for offloading between 2001-2010. While Chow et al., [Chow et al., 2009], write in their paper about the three concerns that are associated with cloud computing, namely: Traditional security, availability and third-party data control. They also propose potential solutions to these concerns by using Trusted Computing and encryption. Anagnostou et al. in their paper, [Anagnostou et al., 2002], writes about different projects in the realm of pervasive computing, the different necessary building blocks for implementing such system and how their project fits into the picture to make pervasive computing a reality.

Offloading

Many works focus on formulating an offloading implementation, with different offloading strategies and assumptions, some of which are presented here.

[Hu et al., 2016] investigated the offloading of computational services to the edge for mobile devices using WiFi and 4G LTE networks. They concluded that substantial benefits can be gained from edge computing for highly interactive mobile applications. Umsonst and Barbosa in [Umsonst and Barbosa, 2024] controls a constrained system over a lossy network where there is a risk of package drops using tube-based MPC control. They also provide a theoretical guarantees about the recursive feasibility and tracking capabilities in case of a disturbance or packet losses. Cuervo et al. in [Cuervo et al., 2010] presented a system called MAUI which optimizes the energy saving of mobile devices by offloading computation dynamically utilizing the intermediate language version of the code base. Chun et al., in their work [Chun et al., 2011], developed a system called CloneCloud, which is a flexible application partitioner and execution runtime. This system enables unmodified mobile applications running in a virtual machine to offload part of their execution from the mobile devices onto device clones operating in a cloud. In [Araújo et al., 2013], Araújo et al. provide a method to do event-based control in wireless control systems to reduce the energy usage. They also tested different communication protocols to solve the control problem. [Luo et al., 2015] discusses the challenges in using small UAVs for disaster sensing due to limited computing and energy resources. To address this, a cloud-supported framework is proposed, integrating video acquisition, data scheduling, and offloading for efficient real-time processing.

Offloading decision

When to offload is a very relevant question which some people have focused on. In this thesis a simple offloading decision was used but this could be improved by utilizing a more sophisticated algorithms.

[Gu et al., 2003] focused on implementing an offloading decision mechanism, e.i., when to offload and application partitioning policy, e.i., what to offload. Chen in [Chen, 2014] proposes a game theory-based approach for efficient computation offloading in mobile cloud computing. It frames the decision-making problem among users as a decentralized game, ensuring a Nash equilibrium. A decentralized mechanism is designed to achieve this equilibrium and is shown to be effective and scalable through numerical results.

Infrastructure

Offloading could be archived using different technologies and infrastructures. The following works focused on different aspects of that.

[Habak et al., 2015] introduces the femtocloud system, which dynamically configures multiple mobile devices into a coordinated cloud computing service, suitable for various scenarios such as public transit, classrooms, or coffee

shops. The system architecture is designed to accommodate device churn and ensure seamless operation. A prototype and simulations are used to evaluate the system's performance, demonstrating its efficiency in leveraging available compute capacity. While [Barbera et al., 2013] explores the costs associated with communication between devices and the cloud, focusing on mobile computation offloading and data backup scenarios. The study evaluates the feasibility and costs of these processes using real devices and cloud-based clones, highlighting bandwidth and energy consumption considerations. In [Cheng et al., 2014] the authors discuss using WiFi offloading in vehicular communication to address increasing mobile data demand. It highlights challenges like high mobility and fluctuating channels, examines drive-thru Internet access, reviews current solutions, and suggests future research directions.Park et al., [Park et al., 2017], reviews wireless network design for Wireless Networked Control Systems (WNCS), covering critical interactive variables, control system performance, parameter adaptation, and state-of-the-art approaches. It also identifies research issues and future directions.

WebAssembly

Some of the works that use WebAssembly in their offloading implementation are presented in this sub-section.

Thesis of [Hansson, 2021] explores using WebAssembly for computational offloading at the Edge to improve program performance by reducing execution time and energy consumption. A proof-of-concept system is developed and evaluated across different use cases on Raspberry Pi devices, comparing local and offloaded execution, with native executions as baselines. In the paper written by [Li et al., 2021] the authors uses WiProg to simplify IoT application development by enabling programming for device, edge, and cloud sides in a single language using WebAssembly. It employs an edge-centric approach with automatic processing and compilation, allowing for efficient runtime execution through dynamic code offloading. While [Nurul-Hoque and Harras, 2021] implemented so called Nomad, an interpreter-based environment for running WebAssembly, capable of live-migrating across operating systems and hardware architectures. Evaluation results show that migration, even cross-platform, adds minimal overhead to performance and delays.

2

Controller Offloading

There are many reasons for executing an application or parts of it in the cloud or at the edge. The most obvious one is the access to the compute power that the cloud provides. However, there are also other reasons, e.g., access to large storage capacity, access to additional information or sensor data that can be used to improve the performance, or the possibility to learn from multiple similar applications that are executing in parallel, e.g., identical control loops.

For control applications, however, it is often the compute power that is the focus. This is also the case in this thesis. More compute power means that it is possible to solve larger control problems or solve the same problem more often and/or faster. The controller type that is considered here is Model-Predictive Control (MPC), where a quadratic optimization problem is solved every sampling period. This can be quite time-consuming if a standard off-the-shelf optimization solver is used. Moreover, the amount of time it takes varies from sample to sample. An assumption that one, sometimes implicitly, make then is that it is simply not possible to execute the controller in the local device, at least not at the desired frequency. This may be true if one uses a small micro-controller as local device. However, in our case the local device is a modern Linux laptop which has almost the same capacity as the servers in the edge node. Hence, although it would be possible to execute the MPC at this particular local device, we pretend here that we have a much less powerful local device.

Offloading a controller can potentially give better control performance due to the increased compute power. However, this may be counteracted by the increased latency caused by the communication between the local device and the cloud or edge. Hence, there is often a sweet spot where the control performance is maximized. Offloading the controller to a potentially very powerful cloud data center that is far away will decrease the performance and executing the controller in the local device, if at all possible, will also decrease the performance compared to the performance at the sweet spot. The sweet spot can in many cases be the edge node as shown in Figure 2.1.

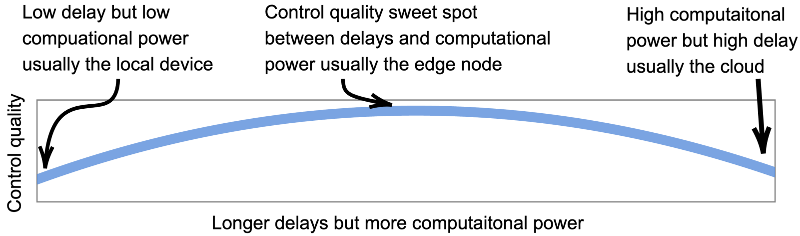

Figure 2.1 Shows an illustrative figure of the sweet spot between high computational power but longer delays and the control quality.

Whenever a controller is offloaded there is always the risk that no control action is returned to the local device. There are several reasons for this. An optimization-based controller may fail to obtain a feasible solution, i.e., a solution that meets the optimization constraints. A wireless connection link can lose the packet being transmitted. Hence, there is a need to have a fallback controller in the local device to switch to if the control signal is not returned within a certain deadline. This logic can be illustrated using a flow diagram as shown in Figure 2.2. In the thesis the fallback controller is a LQR controller, i.e., a state feedback controller that only requires a few multiplications and additions to implement, and which without any problem can execute also on a very resource-constrained local device. The actual switching can be done in several ways. In the MPC case there is also the possibility to use the control signal calculated at the previous control instant. When the MPC optimization algorithm is invoked based on process measurements available at time k it does not only calculate the corresponding control signal but also the control signal to be actuated at subsequent control instants, i.e., at time $k+1$, $k+2$, etc.

A major advantage of MPC is the possibility for the MPC to generate control signals that respect constraints on the process' states and the control signal. Without constraints the MPC in fact is quite close to a LQR controller. In our case the process that is controlled is an inverted Furuta pendulum, i.e., a rotating inverted pendulum. The objective is to balance the pendulum in the upward position. Control of the pendulum in the downward position and control-based swing-up of the pendulum has not been included. For the pendulum balancing problem, control and state constraints are not so natural. One may say that it is difficult enough to balance the pendulum also without constraints. Hence, in our case the difference in control performance between the offloaded MPC and the LQR in the local device is not very large.

The focus of this thesis is dynamic offloading, i.e., the control applications

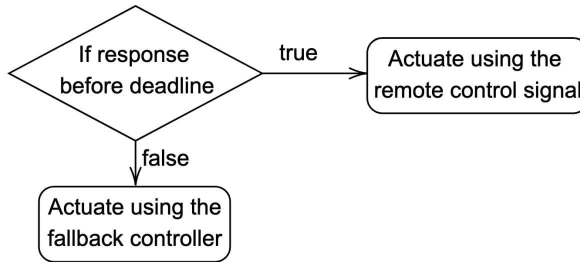

Figure 2.2 Display the offloading logic flowchart, including a fallback action for the local device if no control signal is received before deadline during offloading.

can be moved from one node to another at run-time. That is the reason why WebAssembly (Wasm) is used. Wasm is based on a binary byte code format. As long as an Wasm interpreter or run-time is available in all nodes it is, in principle, possible to move the code to any node. However, transmitting the code over, e.g., 5G takes time and in that case one have to use the local fallback controller to control the process during the transmission as illustrated in Figure 2.3, where the local device preforms the I/O of the process and and the remote solver is the offloading target. There are of course alternative solutions. Instead of actually transmitting the Wasm code one could transmit the URL where the code resides.

A key issue in an offloading strategy is how to decide if to offload, when to offload, and where to offload. In this thesis we assume that we, whenever the control quality is good, want to execute the MPC solver at the edge and that we only have one edge node. Hence, during initialization the code is transmitted to this edge. We also assume that there always is enough compute resources available at the edge node. In a more elaborate scenario one would have to include a monitoring agent that monitors state variables such as the current control performance, and the utilization both at the local device and at the edge, and uses this information to decide if, when and where to offload. In the thesis an approach for monitoring the control performance is proposed.

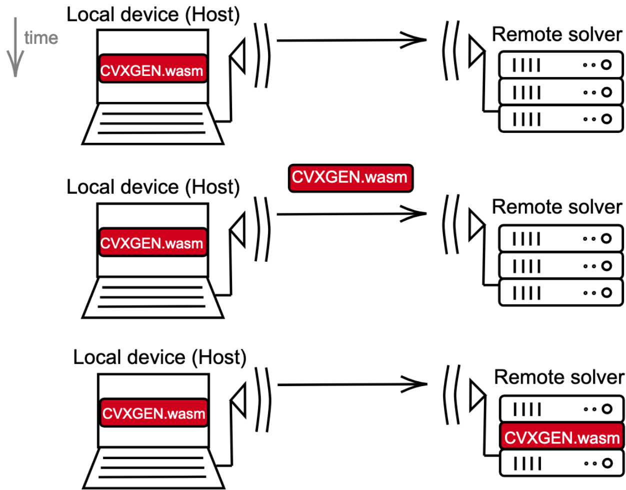

Figure 2.3 A conceptual image of the dynamic offloading implemented in this thesis. The control application, CVXGEN, written in AOT WebAssembly is transmitted to a different node at run-time while the local device is controlling the process using a local controller.

3

Background

The different building blocks used in this work are presented in this chapter. The chapter does not aim to be exhaustive on the different subjects but more of an introductory.

3.1 Furuta Pendulum

The Furuta pendulum, as illustrated in Figure 3.1, is a type of an inverted pendulum used in control theory and robotics . Unlike a traditional inverted pendulum that moves in a single plane, the Furuta pendulum is a two-link system where one controls the base in order to balance the pendulum attached to it [Furuta et al., 1992]. It is characterized by the angles θ and ϕ, representing the pendulum angle and base angle, respectively. The angular velocities associated with these angles are denoted as $\dot{\theta}$ and $\dot{\phi}$.

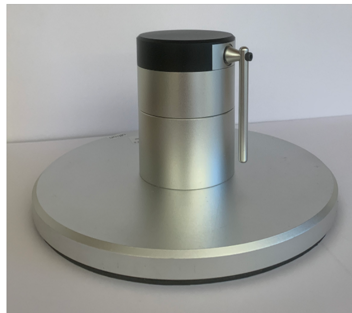

Figure 3.1 The small desktop Furuta pendulum used in this thesis.

The pendulum can be controlled either in the downward position or the upward position. The swing-up of the pendulum from the down to the up

position can be done either in closed loop. i.e., using a controller, or in open-loop using a predefined control sequence. In this thesis it is only the control in the upward position, i.e., the balancing, that is considered. It is assumed that a human user holds the pendulum in the correct position when the controller is started.

The theoretical model of the Furuta pendulum in this work was derived by Gäfvert, [Gäfvert, 1998], using Euler-Lagrange equations. The system of equations derived are presented in Equation 3.1. This non-linear model is valid for all states.

$$
\begin{aligned}
&\frac{d}{dt}\theta = \dot{\theta} \\
&\frac{d}{dt}\dot{\theta} = \frac{1}{\alpha\beta - \gamma^2 + (\beta^2 + \gamma^2)\sin^2(\theta)} \Big(\beta(\alpha + \beta\sin^2(\theta))\cos(\theta)\sin(\theta)\dot{\phi}^2 \\
&\quad + 2\beta\gamma(1 - \sin^2(\theta))\sin(\theta)\dot{\phi}\dot{\theta} - \gamma^2\cos(\theta)\sin(\theta)\dot{\theta}^2 + \delta(\alpha + \beta\sin^2(\theta))\sin(\theta) \\
&\quad - \gamma\cos(\theta)\tau_u \Big) \\
&\frac{d}{dt}\phi = \dot{\phi} \\
&\frac{d}{dt}\dot{\phi} = \frac{1}{\alpha\beta - \gamma^2 + (\beta^2 + \gamma^2)\sin^2(\theta)} \Big(\beta\alpha(\sin^2(\theta) - 1)\sin(\theta)\dot{\phi}^2 \\
&\quad - 2\beta^2\cos(\theta)\sin(\theta)\dot{\phi}\dot{\theta} + \beta\gamma\sin(\theta)\dot{\theta}^2 - \gamma\delta\cos(\theta)\sin(\theta) + \beta\tau_u \Big)
\end{aligned}
\tag{3.1}
$$

The constants α, β, γ, and δ are introduced to simplify the expression. These constants can be calculated using measurements of the pendulum, such as mass and length or be identified though a parameter identification experiment as outlined in Section 3.1.

3.2 Model Predictive Control

Model Predictive Control (MPC) is an advanced control strategy which solves an online optimization problem every control instance to obtain a control signal. This control signal both respects constraints of the process states, the control signal and minimizes the cost defined by the problem formulation [Rawlings et al., 2017].

Mathematical Definition

Consider a discrete-time stabilizable system with dynamics given by

$$x_{k+1} = \Phi x_k + \Gamma u_k,$$

where x_k is the state at time k, u_k is the control input at time k, Φ is the state matrix, and Γ is the input matrix.

The objective of the MPC is to minimize the cost function given by Equation 3.2 over a finite horizon N and not violate the constraints given by Equations 3.3, 3.4 and 3.5 by choosing a sequence of control signals u_i given the initial state x_0.

$$J = \sum_{i=0}^{N-1} \left(x_{k+i}^T Q x_{k+i} + u_{k+i}^T R u_{k+i} \right) + x_{k+N}^T Q_f x_{k+N} \tag{3.2}$$

$$x_{k+i+1} = \Phi x_{k+i} + \Gamma u_{k+i} \quad \text{for } i = 0, 1, \ldots, N-1 \tag{3.3}$$

$$|u_{k+i}| \leq u_{\max} \quad \text{for } i = 0, 1, \ldots, N-1 \tag{3.4}$$

$$\|u_{k+i+1} - u_{k+i}\|_\infty \leq S \quad \text{for } i = 0, 1, \ldots, N-2 \tag{3.5}$$

Q and R are positive definite weighting (cost) matrices, Q_f is the terminal cost matrix, $u_{\max}$ upper and lower control signal constraints and S is the upper constraint on the control signal change between two consecutive time steps. This results in the MCP formulation given by Equation 3.6 [Rawlings et al., 2017].

$$\min_{\{u_j\}_k^{N-1}} \sum_{i=0}^{N-1} \left(x_{k+i}^T Q x_{k+i} + u_{k+i}^T R u_{k+i} \right) + x_{k+N}^T Q_f x_{k+N} \tag{3.6}$$

$$\begin{aligned} \text{subject to} \quad & x_{k+i+1} = \Phi x_{k+i} + \Gamma u_{k+i} && \text{for } i = 0, 1, \ldots, N-1 \\ & |u_{k+i}| \leq u_{\max} && \text{for } i = 0, 1, \ldots, N-1 \\ & \|u_{k+i+1} - u_{k+i}\|_\infty \leq S && \text{for } i = 0, 1, \ldots, N-2 \end{aligned}$$

The variable notation presented in this section will be used throughout this work.

MPC solver

The MPC optimization problem presented above is a convex quadratic program (QP) problem since the model is linear, the constraints are polyhedral and the cost is quadratic. The solver used in this work was CVXGEN. CVXGEN automatically generates a fast solver written in C given a small QP problem formulation[Mattingley and Boyd, 2012]. More information about CVXGEN concerning the problem formulation and a code example are presented in Section 4.4.

3.3 Linear Quadratic Regulator

The Linear Quadratic Regulator (LQR) is also an optimal controller. The difference between it and the MPC controller is:

- The LQR does not require solving an online optimization problem every control instance.
- LQR has an analytic solution utilizing the discrete-time (algebraic) Riccati equation.
- The states and controls signal cannot be constrained [Kwakernaak and Sivan, 1972].

It is interesting to note that if the MPC had an infinite horizon with no constraints the solution would coincide with the solution given by an infinite horizon LQR controller.

Mathematical Definition

Following the same notation introduced in Section 3.2. The objective of the LQR is to minimize the cost function given by Equation 3.7 over a finite prediction and control horizon by choosing a sequence of control signals given the initial state:

$$J = \sum_{i=0}^{N-1} \left(x_{k+i}^T Q x_{k+i} + u_{k+i}^T R u_{k+i} \right) + x_{k+N}^T Q_f x_{k+N} \tag{3.7}$$

Resulting in the LQR formulation given in Equation 3.8 [Kwakernaak and Sivan, 1972].

$$\min_{\{u_j\}_k^{N-1}} \sum_{i=0}^{N-1} \left(x_{k+i}^T Q x_{k+i} + u_{k+i}^T R u_{k+i} \right) + x_{k+N}^T Q_f x_{k+N} \tag{3.8}$$

LQR Solver

The LQR formulation presented in Equation 3.8 could be solved using an analytic solution utilizing either the discrete-time Riccati equation if the horizon is finite or the discrete-time algebraic Riccati equation if the horizon is infinite. The solver returns a state feedback law. If the horizon is short, the state feedback law is time-varying and depending on the initial state, while for infinite horizon, the state feedback law is static and independent of the initial state [Kwakernaak and Sivan, 1972].

There are a few solvers that could be used to derive an analytic solution.

The one used in this work was the dlqr function provided by Matlab. dlqr returns the state feedback law for the infinite horizon problem [MathWorks, 2023].

3.4 WebAssembly

WebAssembly (Wasm) is a binary-code format executed by a stack-based virtual machine [WebAssembly, 2023]. Initially designed for modern web browsers, WebAssembly byte-code is obtained by letting programmers write code in languages like C, C++, and Rust and then compile it to Wasm. The resulting binary-code modules can be accessed from a JavaScript app using the WebAssembly JavaScript APIs [MDN, 2023]. Notably, WebAssembly does not assume any web-specific features, making it applicable beyond the web when an appropriate API is defined [WebAssembly Community Group, 2023].

There exist a WebAssembly Text Format (WAT) which is a human readable format of Wasm binary-code. This format could be used by programmers to write WAT directly and compile it into binary-code but could also be used for programmers to view Wasm modules.

```
1 (func $add (param $a i32) (param $b i32) (result i32)
2   get_local $a
3   get_local $b
4   i32.add)
5 (export "add" (func $add))
```

Listing 3.1 WebAssembly Text Format code example

The code in Listing 3.1 defines a function called "add", denoted by (func $add ...), that takes two parameters, $a and $b, both of type i32, which stands for a 32-bit integer. Inside the function, it retrieves the values of these parameters using the get_local instruction and then adds them together using the i32.add instruction. Finally, it returns the result, which is also of type i32. Additionally, the moudle exports this function so that it can be accessed from outside the module using (export "add" (func $add)).

WasmEdge

WasmEdge serves as a Wasm runtime. It could be used for executing Wasm code in cloud-native environments, edge applications or integrated into programs written in languages such as C, Rust, or Go. WasmEdge runs Wasm code in a sandbox environment to ensure a secure execution environment independent of the underlying operating system[*WasmEdge Documentation Overview* 2023].

In addition, WasmEdge features a compiler that translates WebAssembly into native machine code. This capability enables WasmEdge to operate in Ahead-of-Time (AOT) mode, leading to improved execution speed [WasmEdge, 2023b].

3.5 Containers

Containers are a virtualization technology that encapsulates executable software and enabling it to run inside of the container by invoking system calls to the underlying operating system (OS) for the executable software. Each container possesses its own file system, process space, and network stacks, which are isolated from the hosting OS. Typically, containers are not executed as root users; thus, commands requested by the container may be denied if the container's permissions are insufficient. A container can package all necessary software and dependencies required to execute a specific application, allowing it to be exported as a single unit and run on other nodes, provided that the system calls made by the container are supported [IBM, 2023].

Containers provide an alternative to virtual machines (VMs) by eliminating the need for a hypervisor. This allows for efficient utilization of the features and resources of the host OS [IBM, 2023]. However, containers are usually deployed inside of VMs and these two technologies work together rather than compete.

Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. Kubernetes provides a powerful and flexible infrastructure for deploying and managing distributed systems by abstracting the underlying infrastructure, making it easier to manage complex, multi-container applications [Kubernetes, 2023b][Kubernetes, 2023c].

3.6 5G

5G is the fifth generation of cellular networks which is said to be up to 100 times faster than 4G, have a low latency and have a greater throughput capacity [Ericsson, 2023a]. 5G wireless communication system is expected to support a broad range of newly emerging applications especially since ultra-reliable and low-latency communications (URLLC) is being introduced. URLLC is one of the standout features of 5G technology. It is designed to ensure ultra-reliable and low-latency communication, especially crucial for

mission-critical applications like factory automation, autonomous driving, and virtual/augmented reality experiences. URLLC guarantees good quality of service (QoS), offering low latency of 1 millisecond (ms) along with high reliability [Inseego, 2023]. URLLC was however not used in this work since we did not have the equipment for it.

3.7 Edge Computing

Edge computing, which is a distributed framework which brings processing and storage resources for applications closer to where data is generated and/or consumed, opens up the opportunity for cloud applications to be more responsive and faster to response. This can make it possible to control mission-critical systems which require fast sampling [Ericsson, 2023b].

4

Implementation

This chapter contains an overview of how the experimental setup looked like, a conceptual overview of how the implementation works, a detailed description of the five different components making up this implementation: Model, CVXGEN Solver, Control Quality Measure, Local Device, and Remote Solver. The chapter is concluded by the software overview and in Appendix A the code is provided.

4.1 Experimental Setup

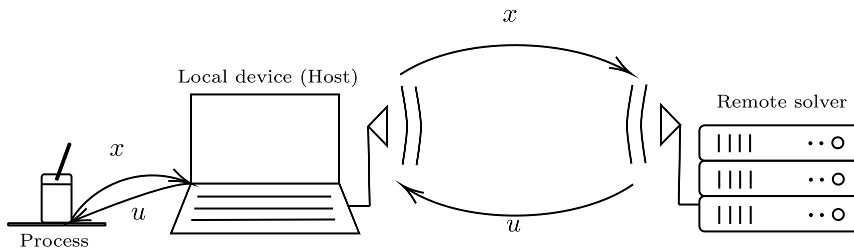

Figure 4.1 Simplified experimental setup overview.

Figure 4.1 provides a simplified overview of the experimental setup. The local device, also referred to as the host, is the computer that controls the pendulum and preforms the offloading. While the remote solver is either a lab-computer or an edge node that preforms the offloaded algorithm. The hardware used in this thesis are:

Furuta Pendulum. A mechanical process developed by Ben Katz and modified by the Department of Automatic Control at LTH. It takes one input (control signal), u, which is the base motor voltage and is proportional to the torque on the base. The system returns also 4 states, the base angle ϕ and

its time derivative $\dot{\phi}$, the pendulum angle θ and its time derivative $\dot{\theta}$ [Pigot, 2021]. To interact with the process the Moberg API was used. Moberg is for connecting and communicating with various lab processes in the Department of Automatic Control at LTH [Blomdell, 2019].

Local device (Host). The local device is a Lenovo T15 computer with a Intel Core i7-10510U CPU @ 1.80GHz with a RAM of 16GB running Fedora Linux 38.

Remote solver. The remote system was either a lab-computer or an edge node. The hardware of the lab-computer also called Heron was *Intel Core i5-4590 CPU @ 3.30GHz* and two 8Gb RAM. While the edge node have the hardware presented in 4.1.

Table 4.1 Edge Node Hardware Information

Node	CPU	# CPUs	RAM
1	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz	20	15.8G
2	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz	20	15.8G
3	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz	20	15.8G
4	Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz	16	15.8G
5	Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz	16	15.8G
6	Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz	16	15.8G
7	Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz	16	15.8G

5G router. The 5G router used to communicate with the edge from the local device was an Askey 5G Sub6 with model name NDQ1300-RoHS dongel.

Wi-Fi 6 router. The Wi-Fi 6 router used to communicate with the edge from the local device was an Archer Ax72 AX5400 Wi-Fi 6 Router.

4.2 Conceptual overview

To understand how the offloaded implementation works an overview of how it run is presented in this section. The details are permitted in this section to make the basic idea clear. The details are presented in the sections that follows.

Upon start the remote solver does not have the control application, the control algorithm, it should use to preform the any offloading tasks, it simply waits for the local device to send the control application. In the same time the local device controls the process, using its local LQR controller and sends the control application to the remote solver as illustrated in Figure 4.2

When the Remote solver receives the control application, it becomes ready to receive offloading requests.

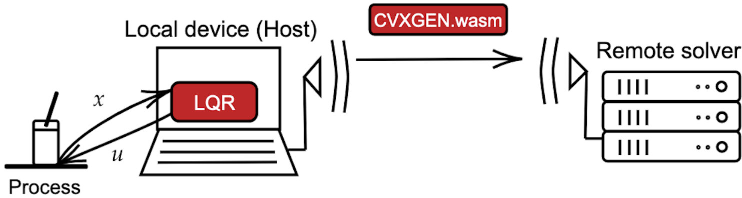

Figure 4.2 An overview on what the program does in the start up phase. CVXGEN.wasm is the control application.

After sending the control application the local device will monitor the control quality of the pendulum. If the control quality is deemed low the local device will control the process using the local LQR controller. Otherwise, if the control quality is deemed good the local device will offload. When remote solver receives an offloading request it uses the control application, received previously, to calculate a control signal and sends it back to the local device. If the response from the remote solver is revived by the local device within a deadline, defined in the local device, the local device will actuate the process with that control signal as illustrated in Figure 4.3. Otherwise, if the deadline is missed the local device will default to the local LQR controller as illustrated in Figure 4.3. This offloading logic is then repeated until the program is stopped.

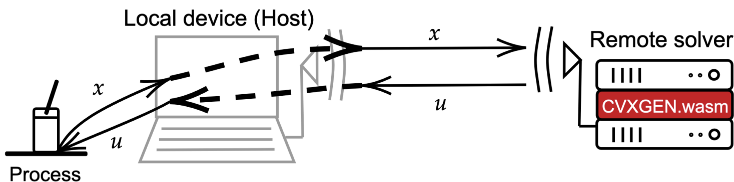

Figure 4.3 An overview of how the program acts if the local device choose to offload and gets a response within the deadline defined in the local device. Here, x is the states of the process and u is the control signal for that state.

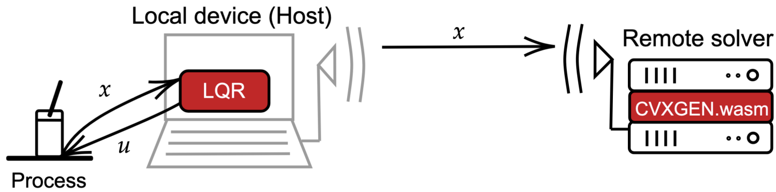

Figure 4.4 An overview of how the program acts if the local device choose to offload and do not get a response within the deadline defined in the local device.

4.3 Model

The control application mentioned in Section 4.2 is at its core a MPC optimization solver. Since a MPC controller is used, a good model of the process is necessary to control it. In Section 3.1, the theoretical model of the Furuta pendulum is presented. The parameters, α, β, γ and δ presented in Equation 3.1 were identified using a parameter identification experiment rather than using the theoretical expressions of these parameters since it gave a better model in this case.

The parameter identification experiment involved exiting the system with a chirp signal, starting with an initial frequency of 0.5 Hz and reaching 12 Hz, with an amplitude of 0.4 Volts and a sampling frequency of 4 ms as shown in Figure 4.5.

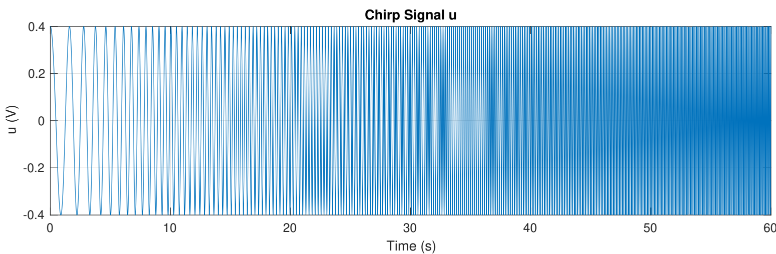

Figure 4.5 Shows the chirp signal used for the model identification experiment.

The parameter identification experiment made on the pendulum while it

was on the down position, i.e., θ is around $-\pi$ as can be seen from Figure 4.6. Since the non-linear model is valid for all states the parameters identified could be used for the upright position without any modification.

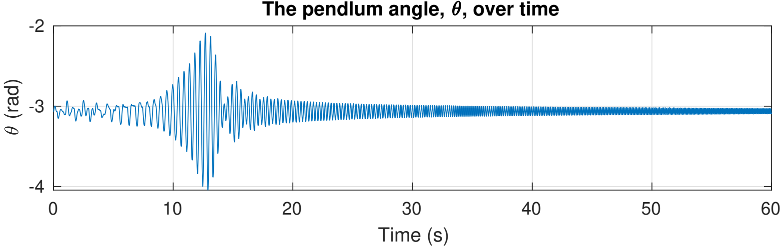

Figure 4.6 Shows the pendulum angle during the model identification experiment.

In the experiment $\theta, \dot{\theta}, \phi, \dot{\phi}$ were recorded and the least square method presented in [Gäfvert, 1998] was performed to get the requested parameters. One can rewrite the system of equations presented in Equation 3.1 as a torque formulation as presented in Equation 4.1.

$$\underbrace{\begin{bmatrix} \tau_u \\ 0 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} \ddot{\phi} & \sin^2\theta\ddot{\phi} + 2\cos\theta\sin\theta\dot{\phi}\dot{\theta} & \cos\theta\ddot{\theta} - \sin\theta\dot{\theta}^2 & 0 \\ 0 & \ddot{\theta} - \cos\theta\sin\theta\dot{\phi}^2 & \cos\theta\ddot{\phi} & -sin\theta \end{bmatrix}}_{\Phi^T} \underbrace{\begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}}_{\Theta} \tag{4.1}$$

where τ_u is the torque applied to the base of the pendulum while the second element in the y vector is the torque applied to the pendulum itself which in our case is 0. $\ddot{\theta}$ and $\ddot{\phi}$ are the accelerations of the pendulum angle and the base angle respectively. It is important to note that the relationship between the torque and the voltage input to the pendulum is unknown. It was assumed that it is a linear proportionality between the voltage input and the torque, which in that case will result in a scaling of our parameters and does not effect the overall model.

To perform the least square, the following steps where followed.

1. The states and the corresponding control signals of the model identification experiment where collected.
2. They measured variables where passed though a first-order Butterworth low pass filter.

3. The accelerations, $\ddot{\theta}$ and $\ddot{\phi}$, were approximated using central difference approximation for the internal points and a forward difference and a backward difference for the edge points.

4. y_i and Φ_i^T were constructed for each time point i,

$$y_i = \begin{bmatrix} u_i \\ 0 \end{bmatrix} \; \& \; \Phi_i^T = \begin{bmatrix} \ddot{\phi}_i & \sin^2\theta_i\ddot{\phi}_i + 2\cos\theta_i\sin\theta_i\dot{\phi}_i\dot{\theta}_i & \cos\theta_i\ddot{\theta}_i - \sin\theta_i\dot{\theta}_i^2 & 0 \\ 0 & \ddot{\theta}_i - \cos\theta_i\sin\theta_i\dot{\phi}_i^2 & \cos\theta_i\ddot{\phi}_i & -\sin\theta_i \end{bmatrix}.$$

5. These points, going from time point 0 to N, where then stacked on top of each other forming:

$$\underbrace{\begin{bmatrix} y_0 \\ \vdots \\ y_N \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} \Phi_0^T \\ \vdots \\ \Phi_N^T \end{bmatrix}}_{A} \Theta.$$

6. The least square formulation gave the requested parameters Θ by calculating,

$$\Theta = (A^T A)^{-1} A^T Y.$$

7. The nonlinear model was linearized around the unstable stationary point $x = [0, 0, 0, 0]^T$, and a friction term was introduced to the model resulting in

$$\begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \\ \dot{\phi} \\ \ddot{\phi} \end{bmatrix} = \begin{bmatrix} -f & 1 & 0 & 0 \\ \frac{\alpha\delta}{\alpha\beta-\gamma^2} & 0 & 0 & 0 \\ 0 & 0 & -f & 1 \\ -\frac{\gamma\delta}{\alpha\beta-\gamma^2} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{\gamma}{\alpha\beta-\gamma^2} \\ 0 \\ \frac{\beta}{\alpha\beta-\gamma^2} \end{bmatrix} u_k.$$

8. The identified parameters were substituted into the linear continuous model and then discretized using the zero-order hold method with a sampling period of 11 milliseconds. This decision came after trying different approaches. Initially, controlling the pendulum within a 15-20 ms period seemed best. However, when we tried this range for discretization, we couldn't stabilize the pendulum. We found that using a discretization period of around 11 ms worked better, allowing stabilization even with sampling periods up to 25 ms. This discretization resulted in the model presented in Equation 4.2. Which is the model used throughout this work to stabilise the pendulum in the upright position.

$$\begin{bmatrix} \theta_{k+1} \\ \dot{\theta}_{k+1} \\ \phi_{k+1} \\ \dot{\phi}_{k+1} \end{bmatrix} = \begin{bmatrix} 1.0064 & 0.0110 & 0 & 0 \\ 1.2676 & 1.0070 & 0 & 0 \\ -0.0004 & 0 & 0.9995 & 0.0110 \\ -0.0666 & -0.0004 & 0 & 1 \end{bmatrix} \begin{bmatrix} \theta_k \\ \dot{\theta}_k \\ \phi_k \\ \dot{\phi}_k \end{bmatrix} + \begin{bmatrix} -0.3163 \\ -57.5884 \\ 0.2139 \\ 38.9051 \end{bmatrix} u. \tag{4.2}$$

4.4 CVXGEN

The MPC solver used in this thesis is the CVXGEN solver. The solver was obtained by formulating the problem as illustrated in Figure 4.7 in the CVXGEN website, where, among other things, the number of states, the horizon and the number of control signals where defined. The CVXGEN website then returned an MPC solver written in C.

```
1 dimensions
2   m = 1  # inputs.
3   n = 4  # states.
4   T = 60  # horizon.
5 end
6
7 parameters
8   A (n,n)  # dynamics matrix.
9   B (n,m)  # transfer matrix.
10  Q (n,n) psd  diagonal # state cost.
11  R (m,m) psd  # input cost.
12  x[0] (n)  # initial state.
13  u_max nonnegative  # amplitude limit.
14  S nonnegative  # slew rate limit.
15 end
16
17 variables
18  x[t] (n), t=1..T+1  # state.
19  u[t] (m), t=0..T+1  # input.
20 end
21
22 minimize
23  sum[t=0..T+1](quad(x[t], Q) + quad(u[t], R))
24 subject to
25  x[t+1] == A*x[t] + B*u[t], t=0..T  # dynamics constraints.
26  abs(u[t]) <= u_max, t=0..T+1  # maximum input box constraint.
27  norminf(u[t+1] - u[t]) <= S, t=0..T  # slew rate constraint.
28 end
```

Figure 4.7 Formulation of the problem for CVXGEN. The prediction horizon was 60, the number of states was 4, and the process had one input.

Every time the CVXGEN solver is invoked, one must first define a parameter list consisting of the states of the process ($\mathbf{x}$), the state matrix (Φ), the input matrix (Γ), and the MPC parameters presented in Equation 3.6, such as the cost on the states (Q), the cost on the control signal (R), the maximum and minimum value of the control signal ($u_{\max}$), and the maximum rate change of the control signal (S). In this formulation, the final cost (Q_f) was omitted since the horizon was deemed long enough. Since the process and the MPC parameters remain unchanged in our case, they are stored in a list and reused every time the CVXGEN solver is called along with the new states. To compile the solver given by CVXGEN using `gcc -Wall -Os` takes around 13 minutes, (when the horizon is 40 it takes around 5 minutes to compile).

Wasm & WasmEdge

The aim with CVXGEN was to compile it to Wasm and use WasmEdge runtime to run it. To interact with the code in Wasm format, a file named `bridge.c` was implemented. This addition acts as a bridge to the CVXGEN solver. It defines the global structures and creates an interface between the CVXGEN solver and the program that interacts with it.

Subsequently, the source code of the solver and the complementary `bridge.c` where compiled to Wasm using Emcc, which is a compiler toolchain that can Compile C or any other language that uses LLVM, into WebAssembly [Emscripten, 2024].The compilation process using `emcc -Wall -O3` took around 11 min. When testing the Wasm CVXGEN controller on the process it was observed that Wasm in interpreted form was too slow for our use case therefor the CVXGEN Wasm file was further compiled to AOT using the WasmEdge compiler [WasmEdge, 2023b], this compilation took around 30 seconds, resulting in an a faster execution of the solver/controller.

4.5 Control Quality

In order to make an informed decisions on when to offload, it is important to implement a mechanism that takes into account the control quality of the controller and decide on offloading based on that. The underlying logic follows a straightforward principle : If the cost, a measure of the control quality, is lower than a predefined threshold the local device will offload the control task to the remote solver. Else if the cost is higher than the threshold the local device will onload the controlling task and control the process using its local controller as illustrated in Figure 4.8.

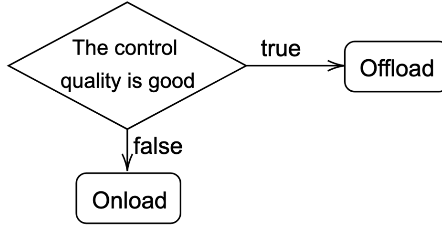

Figure 4.8 The logic used on when to offloading used in this work

Various measures already exist to quantify the control quality of a controller, such as Integral Square Error (ISE), Integral Absolute Error (IAE), Integral Time-weighted Absolute Error (ITAE), and Integral Time Square Error (ITSE) [Schultz and Rideout, 1961]. In the discrete-time case, these measures are defined mathematically as:

$$ISE = \sum_{k=0}^{T} e(k)^2$$

$$IAE = \sum_{k=0}^{T} |e(k)|$$

$$ITAE = \sum_{k=0}^{T} k|e(k)|$$

$$ITSE = \sum_{k=0}^{T} ke(k)^2$$

Here, $e(t) = y(t) - r(t)$, where y(t) is the measured value and r(t) is the reference value at time t. While these control quality measures are widely used, they suffer from a limitation relevant for offloading: they integrate the error over the entire time span. This is suitable for evaluating individual controllers but may not be ideal for our scenario. Implementing this approach would lead to a monotonically increasing cost which will eventually causing the remote controller to be unused.

To address this limitation, a potential solution would involve continuously updating the initial time t_i and final time t_f so that the control quality is a sum over a shorter period, hopefully aligning better with the controller used for pendulum control. However, instead of adopting this approach, we opted to develop our own control quality measure, represented mathematically as:

$$J(k) = 0.5(\theta^2 + \dot{\theta}^2 + \phi^2 + \dot{\phi}^2) + 0.5J(k-1). \tag{4.3}$$

This cost comprises of two components. Firstly, it incorporates the squared deviation of the four states from their reference value (which is zero). Secondly, it includes a fraction of the cost from the previous time. This dual-component structure ensures the cost function's responsiveness to dynamic adjustments while maintaining a degree of stability for noise.

4.6 Local device (Host)

The local device as described in Section 4.2 is the device that is physically connected to the process, reads the process' states and actuates the process. The local device have three functionalities namely:

1. Sending the control application to the remote solver.
2. Monitors the control quality of the process.
3. Controls the process by either offloading or onloading.

These functionalities are realized by 4 Posix threads (pthreads) and written in 6 files, namely: `main.c`, `util.c`, `quality.c`, `regul.c`, `sender.c` and `controller.c`. In this section each functionality is explained together with which files are used. Every shared resource between the different threads are protected by mutexes to ensure mutual execution.

The application sender

When the program is started one thread called `sender_thread` is created. This thread do the following tasks and terminates:

1. Establishes a TCP connection with the remote solver. A TCP connection was used because the data sent by the `sender_thread` is important to be delivered in its entirety to the remote solver which is guaranteed by the TCP connection.
2. Sends the solver file, which in this case is a AOT Wasm compiled CVXGEN solver to the remote solver.
3. Sends the parameters that are needed by the CVXGEN solver but do not change between sampling iterations, which are: The state matrix (Φ), the input matrix (Γ), the cost on the states (Q), the cost on the control signal (R), the maximum and minimum value of the control signal ($u_{\max}$), and the maximum rate change of the control signal (S). The Wasm file is around 1.3 MB while the CVXGEN parameters are 200B.
4. When receiving a verification from the remote solver that everything has been sent successfully it notifies the rest of the program to indicate that the remote solver is ready to receive offloading requests.

The `sender_thread` thread is implemented using `sender.c`. Figure 4.9 shows graphically which files are used by `sender_thread` and how the over all program structure of the local device program looks like.

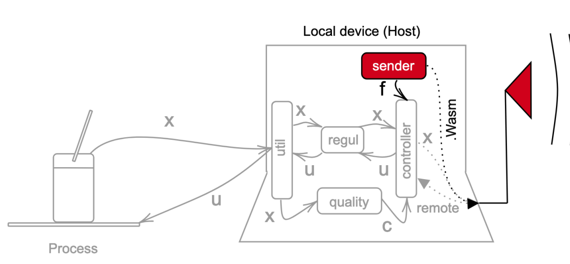

Figure 4.9 Shows an overview of which components that are involved with `sender_thread`, marked with red, which has its primary task of sending the control application and the model parameters to the remote solver.

The control quality monitor

The control quality monitoring is implemented in `control_quality_thread`. The thread does the following tasks with a frequency of 1000 Hz:1)The states of the process are sampled. 2) The measured states and the old control quality measure are used to calculate the new control quality using Equation 4.3.

The `control_quality_thread` thread is implemented in `quality.c` but it uses `util.c`. `util.c` contains functions shared between different threads. These functions include sleep functions and IO-functions. Figure 4.10 shows graphically which files are used by `control_quality_thread`. The sleep function in `util.c` is implemented using the `clock_nanosleep` system call and the `CLOCK_MONOTONIC` clock. The implementation follows Björn Brandenburg's post titled "Liu and Layland and Linux: A Blueprint for 'Proper' Real-Time Tasks" [Brandenburg, 2020] to create a sleep function that sleeps for the correct time period and does not return prematurely.

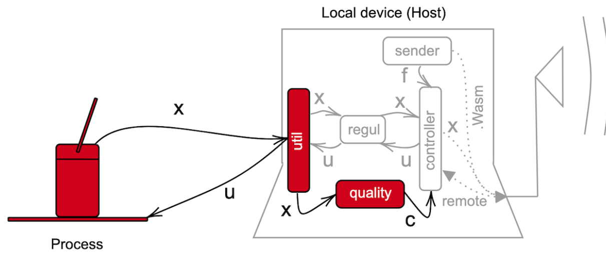

Figure 4.10 Shows an overview of which components that are involved in the `control_quality_thread` thread, marked with red, which has its primary task to measure the control quality using Equation 4.3.

The controller

The controller is implemented in the `regul_thread` thread. This thread does the following tasked with a frequency of 66.67 Hz.

1. Actuate the process with the previously calculated control signal (at time 0 the "previous" control signal is 0.).

2. The states of the process are sampled.

3. If the control application and the model parameters are sent and the control quality is good the thread will offload the control task to the remote solver using a UDP socket. Otherwise, the thread will onload the control task.

4. If the thread chooses to offload it waits for the response from the remote solver. If the the thread get a response within 14 ms it will use the control signal in its next iteration. If the deadline is missed the thread will use the local controller instead to get a control signal.

Figure 4.11 illustrates the offloading flowchart, i.e. step 3 and 4 in the above list.

To clarify the timing, two diagrams showing the two outcomes that could happen when the `regul_thread` thread decides to offload. In Figure 4.12 the remote solver returns a control signal before the timeout deadline of 14 ms while Figure 4.13 shows the timing when that deadline is missed. The blocks in the time line are not drawn to scale and not all details of the program are presented in the figures.

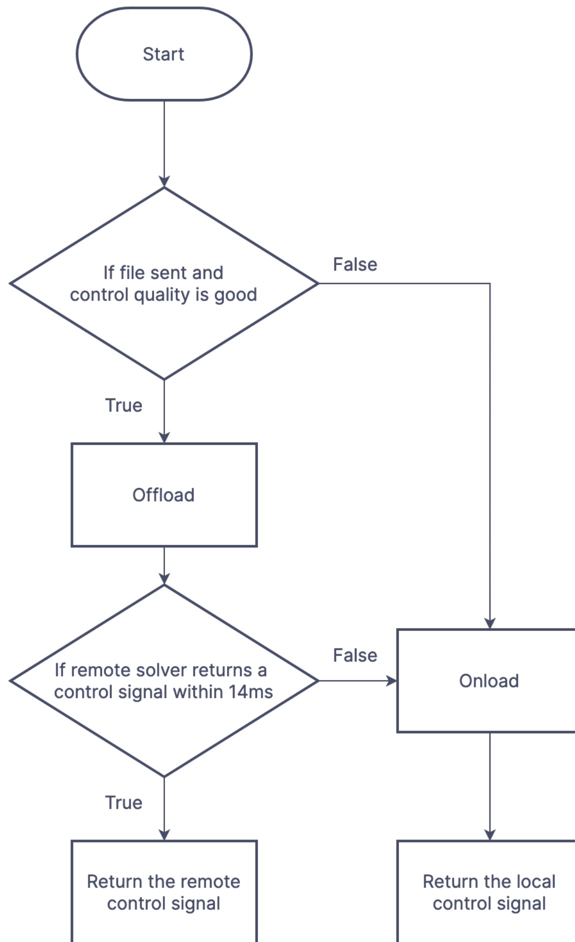

Figure 4.11 Shows the flowchart of the logic used in the `regul_thread` thread on when to offload and when to onload

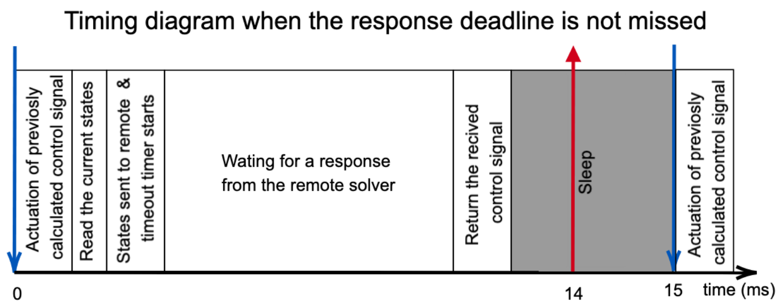

Figure 4.12 Timing diagram of the offloading when the remote solver returns a control signal before the timeout deadline of 14 ms after the states are sent to the remote solver.

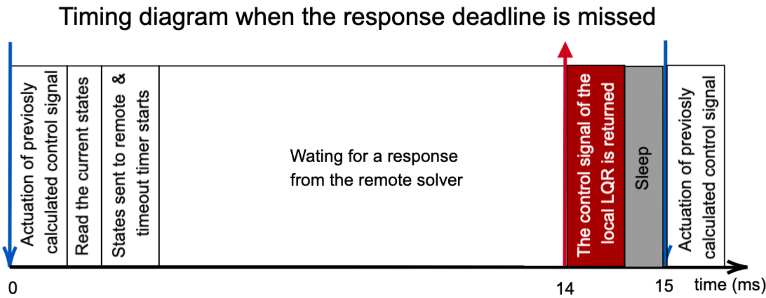

Figure 4.13 Timing diagram of the offloading when the remote solver does not return a control signal before the timeout deadline of 14ms so the local LQR controller is used.

The regul_thread thread is implemented in regul.c but it uses util.c much like the control_quality_thread thread to read the states and to sleep. Unlike control_quality_thread thread the regul_thread thread also uses util.c for actuating the process. The regul_thread thread also uses controller.c file which preforms all the offloading and onloading logic. When regul.c calls controller.c it expects a control signal, it does not know where that control signal comes from.controller.c provide a control signal either from the remote solver of from the local LQR. Figure 4.14 shows graphically which files are used by regul_thread.

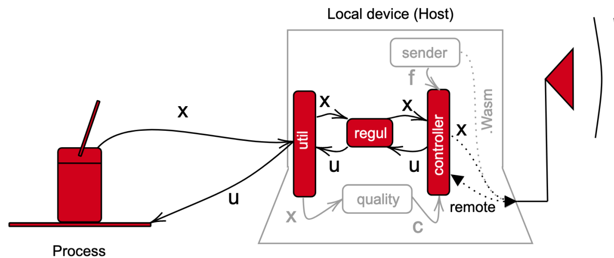

Figure 4.14 Shows an overview of which components that are involved in the regul_thread thread, marked with red, which has its primary task to control the process and preform the offloading.

4.7 Remote Solver

The remote solver is the offloading target for the local device. It starts without the control application it needs to manage offloading requests. The control application is sent to the remote solver when the local device wants to starts offloading. The remote solver has 2 functionalities namely.

1. Receive the control application from the local device using a TCP connection.

2. Use the received control application to calculate the control signal for offloading requests and send it back to the local device using UDP sockets.

These 2 functionalities are implemented using 4 pthreads and three files namely: `main.c`, `receiver.c`, and `main.c`.

The application receiver

When the remote solver program is started a thread called `receiver_thread` is created. This thread blocks the rest of the program until the control application, the AOT Wasm compiled CVXGEN MPC solver, and the CVXGEN parameters, needed to run the CVXGEN solver, are received. The file is saved locally while the parameters are stored on the program memory. The `receiver_thread` is implemented in the `receiver.c`. Figure 4.15 illustrates which files are used to receive the control application.

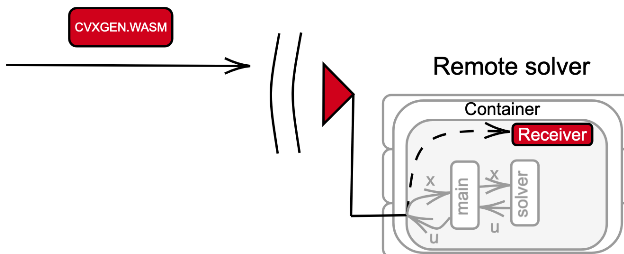

Figure 4.15 Shows an overview of which components that are involved in the `receiver_thread` thread, marked with red, which has its primary task to receiver the control application, CVXGEN.Wasm, and store it locally.

The offloading request handler

When the remote solver program receives the control application it starts two threads namely: `intermediary_thread` and `solver_thread` which together handle offloading requests.

intermediary_thread. This thread acts as the intermediary between the local device and the CVXGEN solver. It opens a UDP socket and starts listening for incoming requested from the local device. When a request from the local device is received it forwards it to the `solver_thread` thread using UNIX sockets. The thread also forwards the response from `solver_thread` thread to the local device. This structure was chosen to asynchronously stop the execution of the solver inside of `solver_thread` when a new request is revived from the local device. Since the execution of the solver is blocking.

solver_thread. This thread invokes the control application and returns the control signal to the requester. It does it by using the WasmEdge runtime to run the CVXGEN function inside of the AOT Wasm compiled file.

The `intermediary_thread` is implemented in the `main.c` file while `solver_thread` is implemented in the `solver.c` . Figure 4.16 illustrates which files to produce the above described functionality.

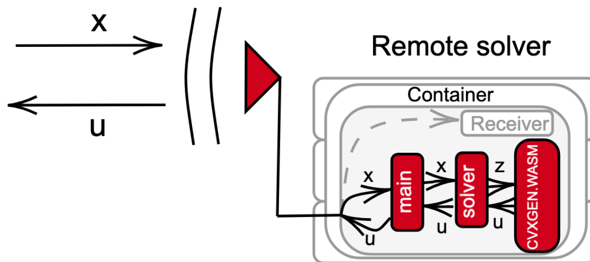

Figure 4.16 Shows an overview of which components that are involved in the `solver_thread` and `intermediary_thread` threads, marked with red, which has its primary task to return a control signal based on the received states using the control application.

The whole remote solver is encapsulated within a container alongside the WasmEdge SDK and the WasmEdge library. The container opens two sockets: one TCP socket for control application reception, and one UDP socket for the offloading requests. The container could be deployed either as a standalone container, by building the image from its container file or by pulling it from the Docker Hub or it could also be deployed inside of

Kubernetes (K8s). To deploy it inside of K8s one can use the `kubectl` CLI, which is a command line tool for communicating with a Kubernetes cluster's control plane [Kubernetes, 2024], and YAML files. In the YAML files one defines a service, a component in K8s responsible for communication between pods and the rest of the system [Kubernetes, 2023d], and a deployment, managing the deployment of containerized application [Kubernetes, 2023a]. When a YAML file is run using `kubectl` CLI, K8s deploys a container that works in the same way as the standalone deployed container.

4.8 Software overview

Figure 4.17 shows a visual representation of the different elements that makes up the offloading implementation presented in this work.

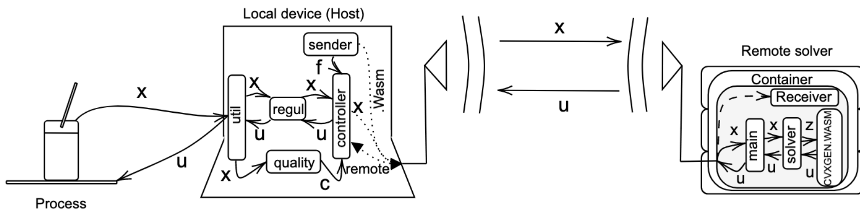

Figure 4.17 Implementation overview that shows the components in both the local device and remote solver.

5

Results

Different types of experiments were conducted to explore different aspects of the offloading framework. The results of these experiments are categorized into four sections: RTT delays for the control signal request, Time for sending the solver file, Control quality and Demonstration.

- **RTT delays for the control signal request.** The total time it took from sending a control request, calculate a control signal, and sending back the control signal to the requester is presented for various configurations.
- **Time for sending the solver file .** The time it took to send the solver file to the remote solver in different configurations is presented.
- **Control Quality.** A presentation of how the different configurations affected the control quality of the process.
- **Demonstration.** Graphs that show the pendulum angle, the control quality measure and which controller is used are presented for when the remote solver is the edge node and the communication medium is the 5G network.

5.1 Timing for the Control Signal

Controlling an unstable system, such as the Furuta pendulum requires short delays. This section analyses delays across different configurations, classifying them into three categories: Integrated control, Container control, and Kubernetes control. This section is then concluded with an overview table of the average RRT delay time and worst case RRT delay time for the different configurations.

- **Integrated control.** All controllers tested are implemented in the local device without offloading. Forming a baseline to how fast the controllers are.

- **Container control.** In this section the results of the implementation written about in Chapter 4 are presented. Concretely, the offloading implementation composed of the host and a remote solver is tested. In this subsection the remote solver is deployed as a standalone container.

- **Kubernetes control.** The deployments presented in this section are similar to those of the Container control section. The difference being that the remote solver is the edge, and the remote solver container is deployed inside of a Kubernetes cluster.

In Container control and Kubernetes control subsection the RRT delay time is the duration from when `controller.c` gets a control signal request until `controller.c` returns a control signal, as Illustrated by Figure 5.1. While in Integrated control subsection the RRT delay time is simply the execution time of the code since no offloading is preformed.

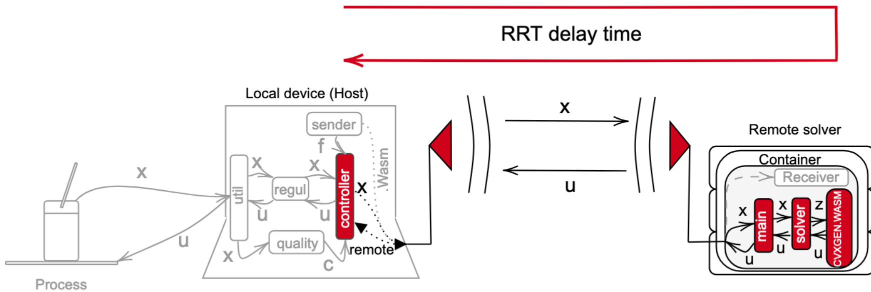

Figure 5.1 Shows visually what is meant by RRT delay time for Container control and Kubernetes control.

Integrated Control

Integrated control refers to the scenario where the pendulum is controlled by the host computer without any offloading. The local device used in this work, as mentioned in Section 4.1, was a *Lenovo T15* computer with am *Intel Core i7-10510U CPU @ 1.80GHz* with a RAM of 16GB running Fedora 38.

In this subsection four controllers are presented: LQR, CVXGEN MPC written in C, Wasm CVXGEN MPC and AOT Wasm CVXGEN MPC. Each is explained in its own subsection. Since no offloading is preformed in this subsection the RRT delay time simply becomes the execution time of the code.

LQR. This controller is noteworthy as it will serves as the local controller, and having a fast fallback controller is essential to ensure stability when a control signal is needed in a short period of time.

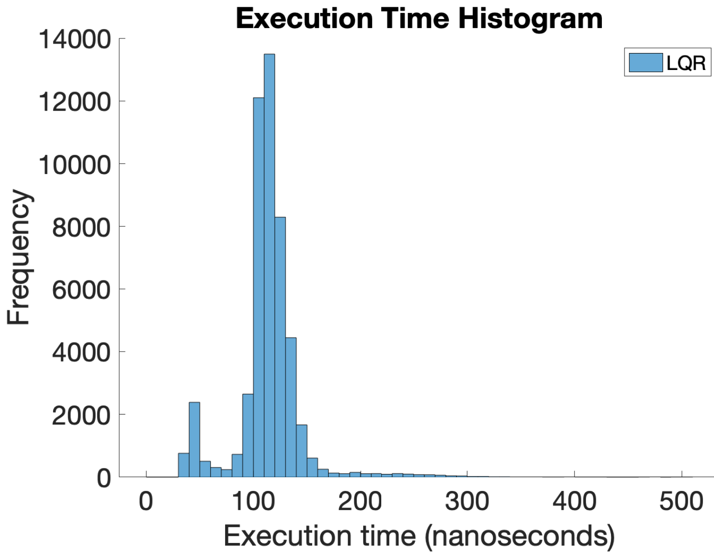

Figure 5.2 Execution time histogram for the LQR controller in nanoseconds

As can be seen from Figure 5.2 the LQR controller is fast, the average execution time is around 120 ns. This performance is expected, given the simplicity of the controller. It can be implemented by one line of code.

MPC C-code. The CVXGEN MPC controller, written in C, is tested to determine the best-case scenario for the MPC solver's execution time.

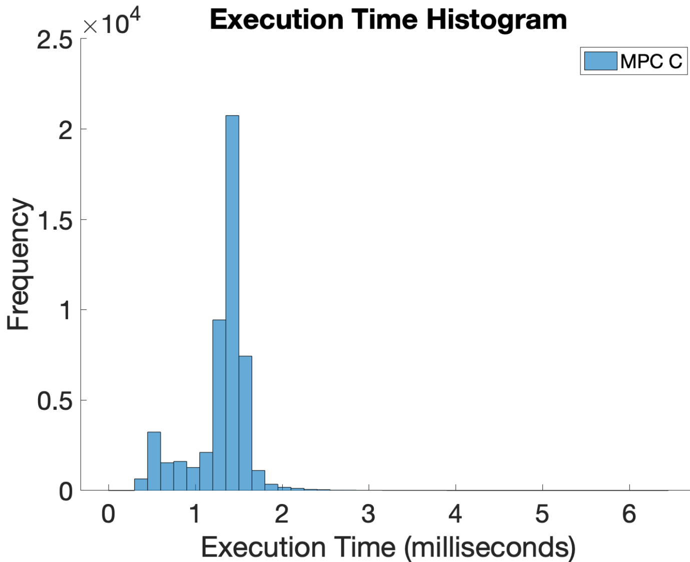

Figure 5.3 Execution time histogram of the MPC controller executed as C-code in miliseconds

The execution time of the MPC solver ran as C-code is presented in Figure 5.3. The average execution time is 1.3 ms, i.e., much longer than for the LQR controller. This difference can be attributed to the fact that the MPC controller must perform a minimization to determine the optimal control signal every time step while LQR is static. The length of the horizon plays a crucial role for the execution time. In this work a horizon of 60 was used, if the horizon were shorter the execution time would decrease. Although some experiments were conducted with controllers with shorter horizons, they proved to be unreliable and no further experiments where done.

MPC Wasm. As mentioned in Section 4.4 the CVXGEN solver, together with `bridge.c`, was compiled to Wasm. The Wasm file was then integrated into the control loop using WasmEdge's runtime. As evident from Figure 5.4, the execution time of the WASM CVXGEN solver in interpreted form is notably slow, averaging 219 ms. While it might be possible to stabilize the pendulum by using a better model and/or a shorter horizon, it will be challenging. For context, the longest period found where it was possible to stabilize the Furuta pendulum was 50 ms using LQR while for MPC the longest period was 25 ms.

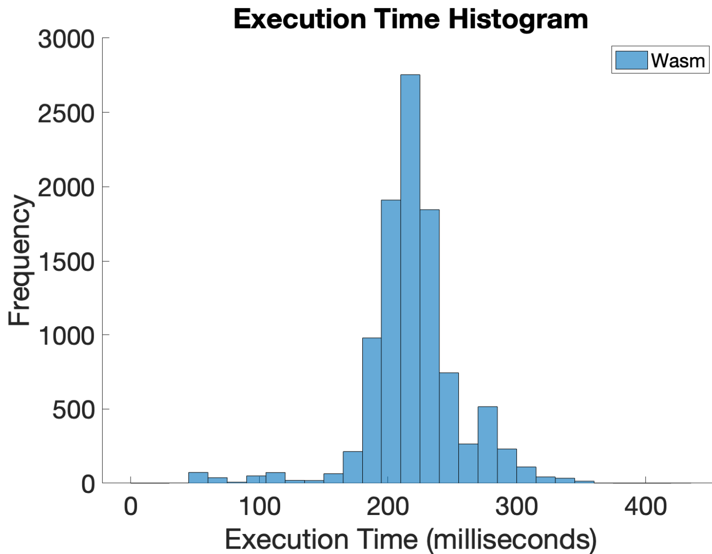

Figure 5.4 Execution time histogram for MPC controller executed as WASM in miliseconds

MPC AOT Wasm. Interpreting Wasm is too slow to stabilise the pendulum. Therefore, Ahead-of-Time (AOT) compilation of Wasm was tested to stabilize the pendulum. The data presented in Figure 5.5 illustrates a notably speed improvements of the AOT Wasm execution compared to interpreted Wasm.

This efficiency gain comes at the cost of increased file size. The AOT code gets integrated into the Wasm file, allowing it to be executed either as interpreted Wasm or AOT-compiled code. The initial size of the Wasm solver is 666 KB, while the AOT Wasm file has a size of 1.3 MB, approximately twice as large. This have two implications. 1) If the local device that controls the system has limited memory the solver might be too big to be stored locally. 2) It will take longer time to send the solver to the remote solver. It takes on average 1.8s sending the AOT-compiled Wasm solver to the edge node using the 5G network.

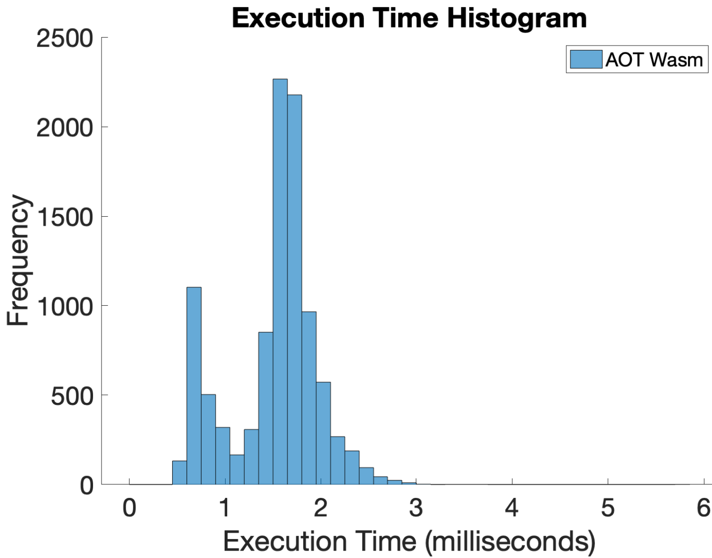

Figure 5.5 Execution time histogram for MPC controller executed as AOT Wasm in miliseconds

Container Control

In this subsection offloading was tested. The remote solver was deployed inside of a container either locally on the local device or remotely on an another computer. Based on the results presented in the Integrated control subsection the LQR controller was chosen as the fallback local controller for its fast execution speed and the AOT-compiled Wasm CVXGEN was used as the remote solver's controller. The reason being that as long as a Wasm interpreter or run-time is available in all nodes it is, in principle, possible to move the code to any node.

The RRT delay time of the two deployments, explained further in their own subsection, are shown in Figure 5.6. In the figure one can also see the threshold of 14 ms is also marked. This threshold is the time `controller.c` wait until it abandons the remote solver for that iteration and uses the local LQR controller instead when offloading.

Local Deployment. The remote solver is deployed in the host computer, meaning that the container is run on the same computer as the local device and the loop-back ip address is given to the local device to use when ”offloading” to the container. This deployment is a form of a baseline on how fast the host to remote solver communication can be. It is evident that the distribution in Figure 5.6 is more spread out and has a longer execution time

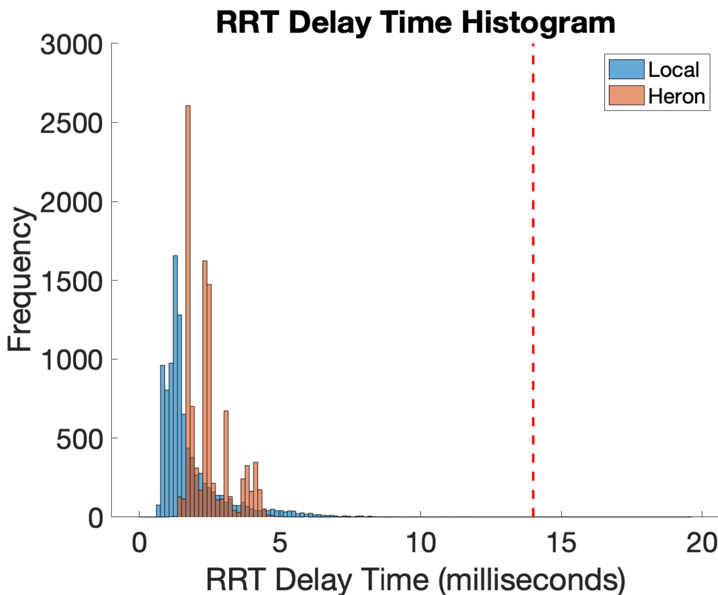

Figure 5.6 RRT delay time histogram for Local and Heron deployments together with the predefined deadline of 14 ms.

compared to Figure 5.5. This spread could be attributed to the fact that the computer has multiple processes running and competing for resources.

Heron Deployment. The remote solver is deployed in another computer, called Heron-01. Heron-01 has an *Intel Core i5-4590 CPU @ 3.30GHz* and 2 *8Gb* RAM sticks. The communication medium between the local device and Heron-01 was Cabled-Ethernet. Similar to Local Deployment, the deployment on Heron exhibits a comparatively slower execution time compared to the integrated AOT Wasm solver but fast enough to stabilize and control the Furuta pendulum.

Kubernetes Control

As previously mentioned, the edge node in the Department of Electrical and Information Technology (EIT) at Lund University is a bare-metal Kubernetes cluster connected to the network. The hardware of the edge node used in this work is presented in Table 4.1. To deploy the remote solver the container had to be run inside of a Kubernetes cluster. This was done by defining services and deployments and running them using kubectl as explained in Section 4.7. These YAML files could be used to deploy the remote solver on any Kubernetes cluster, enabling the offloading to the cloud or other

Kubernetes clusters.

Since the only remote solver deployed inside of K8s was the edge node different communication technologies where tested, namely: 5G, Wi-Fi, and Cabled Ethernet. The connection to the edge was routed though a router, namely: Archer AX72 AX5400 Wi-Fi 6 Router for both the Wi-Fi communication and the cabled Ethernet communication and though the 5G antennas and a core network server, i.e., with no direct connection between the local device and the edge. Another detail to mention is that we did not specify on which server in the edge node that the remote solver was deployed in, it was assumed that all servers where equally fast. The collected data using the edge as the remote solver and testing the different communication technologies is presented in Figure 5.7 together with the deadline of 14 ms mentioned in Section 5.1.

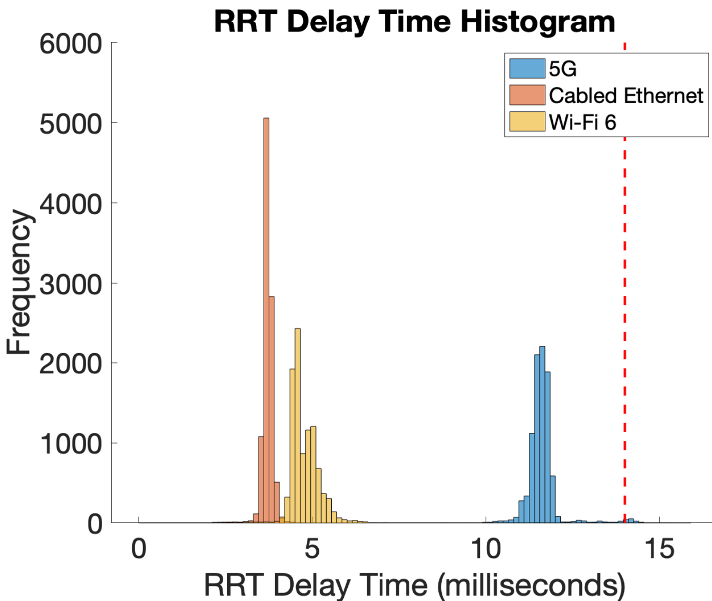

Figure 5.7 RRT delay time histogram for 5G, WiFi and Cabled Ethernet deployment inside of a K8s cluster with the predefined deadline of 14 ms.

5G Communication. As shown in Figure 5.7, the majority of the RRT delay time falls within the deadline defined, making it possible for the local device to get back the majority of sent offloading requests in a timely fashion. On avergae the RRT delay time for the 5G network was 11.6 ms. The pendulum can be controlled over 5G with a maximum period of 22 ms.

Allowing us to increase the deadline threshold even more and thus include all requests before the deadline threshold. We opted not to control it with a longer period because the pendulum was not robust enough with such long delays and to showcase how `controller.c` dynamically switches to the local LQR when the deadline is missed.

In this work a standard 5G network, rather than 5G Ultra-Reliable Low-Latency Communication (URLLC) network was used. This is because we do not have access to URLLC hardware to utilize it. This work, however, demonstrates that a regular 5G network could be leveraged for offloading, even though 5G URLLC is often touted for its low-latency capabilities and most probably will result in a shorter delay allowing for even faster processes to be offloaded over the 5G network.

Additionally, it is worth noting that the delay also depends on the 5G router used on the local device. Three different routers where used are: *Askey 5G Sub6* model name *NDQ1300-RoHS*, *WNC SKM-5xE*, and `RUTX50`. All gave different delays indicating that different routers have different implementations and thus varying speed. None of the 5G routers tested in this thesis had too long delays for stabilizing the pendulum but the latency variation was in milliseconds. We opted to present the *Askey 5G Sub6* 5G router to simplify the presentation.

Wi-Fi Communication. In this work Wi-Fi 6 with a bandwidth of 1Gbps was also investigated for offloading, since WiFi is prevalent and accessible for many industries, making it relevant for scenarios where 5G may not be available or practical. In the experimental setup, WiFi 6 had lower latency compared to 5G with an average of 4.8 ms.

In theory, 5G should provide fast communication, even in crowded scenarios with numerous users, as highlighted by Ericsson [Ericsson, 2023a]. In contrast, WiFi may experience slowdowns as the number of devices increases. This work did not experiment with or test the network for scenarios involving many end devices and justify this claim.

Cabled Ethernet Communication. While wired Ethernet offers high data transfer rates of 1Gbps, the trade-off is a limitation on the flexibility of the local device and the process due to the constraints imposed by the physical cable. In the experimental setup presented in this work, the mobility is not crucial for the control task. If, however, the framework have been tested on a mobile process the cabled constraints would be evident. Cabled Ethernet is the fastest communication technology with the edge node with an average of 3.7 ms.

Overview

The different configurations presented above cloud be summarized more visually using Table 5.1. This naming convention will persist though out this

work.

Table 5.1 Technology implementation for the different configurations.

Name	Remote Solver	Com. Technology	Controller
LQR	none	–	LQR
MPC C-code	none	–	MPC C-code
Wasm	none	–	Wasm MPC
AOT Wasm	none	–	AOT Wasm MPC
Local	Local device (Container)	–	AOT Wasm MPC
Heron	Computer (Container)	Cabled Ethernet	AOT Wasm MPC
5G	Edge node (K8s)	5G network	AOT Wasm MPC
Wifi	Edge node (K8s)	Wi-Fi 6	AOT Wasm MPC
Cabled Ethernet	Edge node (K8s)	Cabled Ethernet	AOT Wasm MPC

The variations in the RRT delay time across different configurations, highlighting both the potential and limitations of various deployment methods and communication technologies. To facilitate comparison and provide a centralized location for presenting data, Table 5.2 presents the average and worst-case execution times for the different configurations, following the naming convention presented in Table 5.1.

Table 5.2 Average and Worst-case execution tor RRT delay times

Scenario	Average Time (ms)	Worst-case Time (ms)
LQR	0.00012	0.032
MPC C-code	1.3	6.4
Wasm	219	424
AOT Wasm	1.5	5.7
Local	1.8	13.1
Heron	2.4	19.6
5G	11.6	15.8
WiFi	4.8	14.2
Ethernet	3.7	8.7

5.2 Timing of the File Sending

As mentioned in Section 4.2, the solver file is sent to the remote solver to achieve flexibility in choosing which control application to use when offloading. In this section the time it takes for the `sender.c` to send the AOT Wasm compiled CVXGEN MPC solver to the remote solver as illustrated by Figure 5.8.

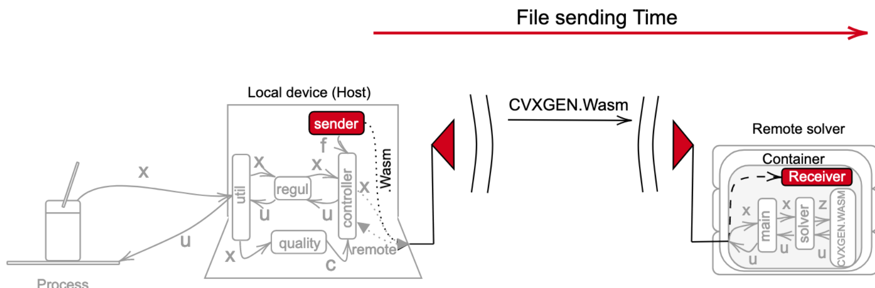

Figure 5.8 Shows visually what is meant by timing of file sending.

In Figure 5.9, the spread of how long it took to send the solver file to the remote solver is illustrated. For each plotbox in the Figure we collected around 10,000 data points. An exception was made for the 5G network where only 1300 data points where collected. It is evident from the plot that sending the file to the container deployed locally is the fastest, and sending it to the edge using the 5G network is the slowest. It could be argued that the solver does not need to be sent to the solver in the local deployments and that is true. This was still done and included in this work to have a baseline

on how fast transmitting the file takes. The average and worst-case times in milliseconds are presented in Table 5.3 following the naming convention presented in Table 5.1.

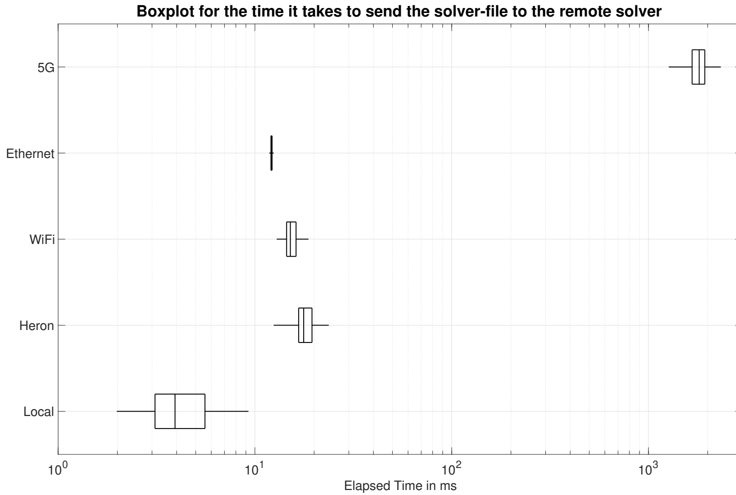

Figure 5.9 The time it took to send the the AOT-Wasm file to the remote solver in milliseconds for the different configurations.

In this implementation, the solver was sent using default TCP sockets (the `TCP_NODELAY` flag was activated). This is sub-optimal since more optimized configurations will send the file faster. The bandwidth of the different communication protocols allowed for more data to be sent per second compared to what is shown here. Another way to increase the speed of transmission is to compress the solver and send a compressed solver and decompress it at the remote solver. This is however not a significant issue since the pendulum is controlled using the local LQR until the file is uploaded, as can be seen from Figure 5.10 and mentioned in Section 4.2.

Table 5.3 Average and Worst-case times of sending the AOT-Wasm file for the different configurations

Scenario	Average Time (ms)	Worst-case Time (ms)
Local	5	491
Hero	22	463
5G	1791	2336
WiFi	16	94
Ethernet	12	265

5.3 Control Quality

As mentioned in Section 4.5 a simple control quality measure is proposed. The local device use this control quality measure to decide if to offload. Table 5.4 presents the control quality of the different configurations and the fraction of times the remote solver controls the process, following the same naming convection as presented in Table 5.1. For each average value presented in Table 5.4, we have collected 10,000 data points.

One can see from Table 5.4 that offloading to the control quality measure seems to fluctuate even if the fraction of offloading is the same and it is the same controller deployed in the remote solver. Indicating that the control quality measure used is too simplistic and random noise is effecting the data. Even though our formulation of the control quality measure aims to decrease the fluctuation of random noise in the measurement the averaging applied was too small to create a smooth measure. A more conservative control quality, normalising the states and including the control signal might result in a better control quality measure.

Table 5.4 Average values of the custom cost for the different configurations

Scenario	**Avg. Control Quality Measure**	**Fraction of offloading**
LQR	4.3	0%
Local	5.22	97.80%
Heron	5.62	97.78%
5G	2.67	91.05%
WiFi	6.41	97.95%
Cabled Ethernet	2.03	97.55%

5.4 Demonstration

In Figure 5.10, we illustrate the pendulum angle, θ, during the dynamic offloading of the pendulum, using 5G to communicate with the edge. Notably, the pendulum assumes the upright position, as indicated by θ being close to zero. From the figure in the middle, it can be seen which controller is used. Initially, the LQR controller controls the pendulum while the solver file is being sent to the edge. The third graph shows control quality measure over time. The red dashed line represents the threshold used to decide if the remote or the local LQR controls the pendulum. This threshold was determined through experimentation; it can be observed that the threshold is violated sometimes, and then the LQR takes over and controls the pendulum. This threshold could be decreased if one desires a more conservative control of the pendulum. It is also interesting to note two things. First, the control quality is more oscillatory with more spikes when the LQR controller is controlling the pendulum, indicating that we achieve better control when offloading compared to controlling it locally. Second, it is evident from the pendulum angel, θ, that the pendulum fluctuates more when controlled using the local LQR compared to the remote MPC which means that the system is controlled better using the edge node compared to the local control.

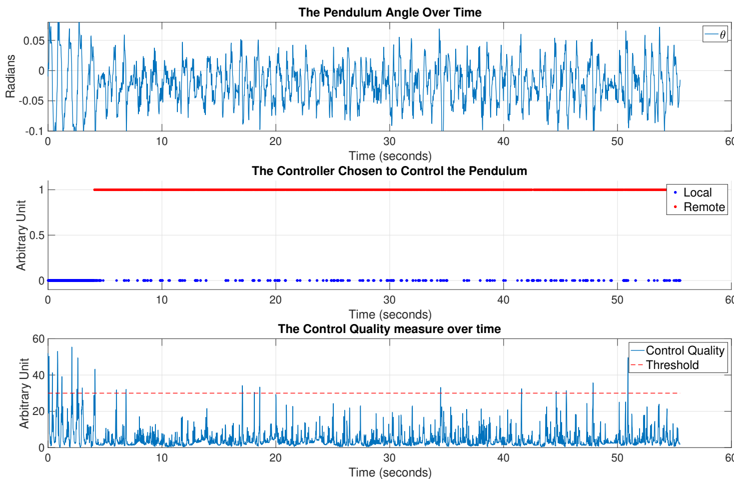

Figure 5.10 The first graph shows the pendulum angle, θ, the second graph shows which controller is used to control the pendulum, and the third graph shows the control quality and the threshold used.

6

Thesis Summary

As can be seen from the work, the offloading implementation did work for a different number of configurations of which some used the edge and/or 5G network while others did not.

Interestingly the computational power of the host is sufficient for both the MPC controller and the simple LQR controller. The aim of this work was not to control a system that can not be controlled by the host computer but to implement an offloading strategy, develop a control quality measure, and to demonstrate its capabilities by controlling a process with fast dynamics.

It was determined that WebAssembly in interpreted form was too slow to control a process with fast dynamics such as the Furuta pendulum, but that AOT-compiled Wasm made it more suitable for this use-case. This work also found that, although WiFi and Wired Ethernet were faster, the delays of offloading to an edge node using a regular 5G network were deemed short enough for offloading, as it could control the pendulum. Finally, it was also noted that the control quality measure used in this work could be improved by introducing weights to the states and by including the control signal in the control quality measure.

6.1 Weaknesses and Potential Criticism

A careful approach was taken in developing this work to ensure its quality, but due to time constraints, some implementation choices could be discussed and potentially viewed as weaknesses. These are as follows:

- **The offloading strategy assumes four states and one control signal.** This could be perceived as a limitation, considering that some systems have more control signals and states. The decision not to implement a dynamic strategy is due to the advanced nature of dynamic programming in C, which requires a substantial period to implement.

- **Cybersecurity:** This work did not emphasize cybersecurity extensively. We developed a simple package ID and a basic ID check to ensure that the solver can verify that the received package corresponds to the one sent. But no encryption, firewall and error correcting code was implemented.

- **Timeout Sockets:** In the C code, the built-in timeout function of a socket was used, where a specified number of milliseconds is stated, and the socket returns if no answer was received in the specified time. The potential problem with this is that the timer in the timeout function is a best-effort timer, where the socket might return before or after the specified time. A solution to this problem is to implement a more reliable timeout mechanism using different threads.

- **No convergence check for the MPC solver:** Right now it is not possible to check if the MPC solver has converged or not. This limitation is imposed by WasmEdge since only one variable is returned from every function call making it hard to send a flag alongside the control signal. This could be fixed by implementing a helper function in the `bridge.c` that can return the flag.

6.2 Future Work & Improvements

To build upon what was presented in this thesis, several avenues for development exist. Below, we outline some ways in which this work could be extended:

- **Use another QP solver:** CVXGEN is a good QP solver, but one downside is that the prediction horizon cannot be changed dynamically. This is of interest if one wants to adjust the sampling period dynamically. Typically, the horizon is expressed in time units, which translates to a number of samples. Therefore, if the sampling period changes, the horizon expressed in the number of samples needs to change accordingly. Currently, to modify the solver's horizon, one must reformulate the problem on the CVXGEN website and recompile it for use. Another solution is to download multiple solvers with different horizons and store them on the host, choosing among these solvers when offloading. Since this is impractical, it might be interesting to try out another QP solver that supports a dynamic horizon.

- **Kalman filter:** The readings of the states from the Furuta pendulum proved, at times, unreliable. Due to measurement noise and crude approximations. Implementing a Kalman filter could mitigate the effects

of these issues, potentially increasing the period during which the pendulum can be controlled. A Kalman filter was implemented but it was not prioritized to fine tune it.

- **Model identification:** A better model would have made the implementation better and more tolerant for longer delays. Results shown by simulations indicated that the model does not represent the system fully, limiting the control of the pendulum. If a better model was used it might have made it possible to control the pendulum over the cloud or using interpreted Wasm instead.

- **Rust implementation:** It would have been interesting to implement the entire program in Rust instead of C, as WasmEdge have implemented more tools and documentation for Rust. Since sockets are supported in Rust, this shift would enable the use of WasmEdge containers instead of a Linux containers [WasmEdge, 2023a].

- **Including the time Delay in the MPC formulation:** In the MPC formulation, the incorporation of a one-sample delay was considered with the aim of enhancing the robustness of the controller. This was implemented and tested but the outcome was unsuccessful in the real process but proved successful in the simulation. One plausible reason for this is that the model does not match the real Furuta pendulum.

- **Horizontal offloading:** In this study, offloading occurred to a single remote solver while the pendulum remained stationary. An intriguing avenue for further exploration is horizontal offloading, where the host can dynamically offload the solver to different edge nodes based on predefined metrics. In such scenarios, the solver could be offloaded to the edge with the optimal measurements, and it could dynamically switch to a different edge if superior performance is indicated by the metrics. This dynamic adaptation opens up possibilities for offloading when the local device is moving.

- **Delay and jitter margin:** In the thesis whether or not the process is stable for a certain sampling period was evaluated experimentally. An alternative would have been to use analytical tools for this. Using the delay margin one can offline valuate how long constant delay that the process can tolerate before becoming unstable [Hägglund, 2021]. The case of jitter in the delay can also be analyzed using the jitter margin tool [Cervin et al., 2004].

- **Dynamic offloading negotiations:** In the thesis we assume that the resources in the edge node are always sufficient for executing the MPC solver. That is not always the case. The edge node may be a

shared resource used by several applications. This opens up the issue of dynamic resource negotiation between the host (local device) and the edge node. It is often possible to reduce the resource requirements of a control application, e.g., by reducing the sampling period. Performing this negotiation requires knowledge of how the control performance depends on the available resources, e.g., the sampling period.

Bibliography

Anagnostou, M., A. Juhola, and E. Sykas (2002). "Context aware services as a step to pervasive computing". In: *Lobster workshop on location based services for accelerating the European-wide deployment of services for the mobile user and worker*, pp. 4–5.

Araújo, J., M. Mazo, A. Anta, P. Tabuada, and K. H. Johansson (2013). "System architectures, protocols and algorithms for aperiodic wireless control systems". *IEEE Transactions on Industrial Informatics* **10**:1, pp. 175–184.

Årzén, K.-E., P. Skarin, W. Tärneberg, and M. Kihl (2018). "Control over the edge cloud - an mpc example". English. In: 1st International Workshop on Trustworthy and Real-time Edge Computing for Cyber-Physical Systems ; Conference date: 11-12-2018 Through 11-12-2018. URL: `https://cps-vo.org/group/TREC4CPS_conference`.

Barbera, M. V., S. Kosta, A. Mei, and J. Stefa (2013). "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing". In: *2013 Proceedings Ieee Infocom*. IEEE, pp. 1285–1293.

Blomdell, A. (2019). *Moberg gitlab repository*. `https://gitlab.control.lth.se/anders_blomdell/moberg`.

Brandenburg, B. (2020). *Liu and layland and linux: a blueprint for "proper" real-time tasks*. Last updated on 05 Sep 2020. URL: `https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/`.

Cervin, A., B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo (2004). "The jitter margin and its application in the design of real-time control systems". English. In: *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications.*

Chen, X. (2014). "Decentralized computation offloading game for mobile cloud computing". *IEEE Transactions on Parallel and Distributed Systems* **26**:4, pp. 974–983.

Cheng, N., N. Lu, N. Zhang, X. S. Shen, and J. W. Mark (2014). “Vehicular wifi offloading: challenges and solutions”. *Vehicular Communications* **1**:1, pp. 13–21.

Chow, R., P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina (2009). “Controlling data in the cloud: outsourcing computation without outsourcing control”. In: *Proceedings of the 2009 ACM workshop on Cloud computing security*, pp. 85–90.

Chun, B.-G., S. Ihm, P. Maniatis, M. Naik, and A. Patti (2011). “Clonecloud: elastic execution between mobile device and cloud”. In: *Proceedings of the sixth conference on Computer systems*, pp. 301–314.

Cuervo, E., A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl (2010). “Maui: making smartphones last longer with code offload”. In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62.

Emscripten (2024). *About emscripten.* Accessed 2024-02-10. Emscripten. URL: `https://emscripten.org/docs/introducing_emscripten/about_emscripten.html`.

Ericsson (2023a). *Ericsson - 5g.* Accessed: 2023-12-18. URL: `https://www.ericsson.com/en/5g`.

Ericsson (2023b). *Ericsson - edge computing.* Accessed: 2023-12-18. URL: `https://www.ericsson.com/en/edge-computing`.

Furuta, K., M. Yamakita, and S. Kobayashi (1992). “Swing-up control of inverted pendulum using pseudo-state feedback”. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* **206**:4, pp. 263–269. DOI: `10.1243/PIME_PROC_1992_206_341_02`.

Gäfvert, M. (1998). *Modelling the Furuta Pendulum.* Tech. rep. Department of Automatic Control, Lund Institute of Technology. ISRN: LUTFD2/TFRT–7574–SE.

Gu, X., K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic (2003). “Adaptive offloading inference for delivering applications in pervasive computing environments”. In: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003.(PerCom 2003).* IEEE, pp. 107–114.

Habak, K., M. Ammar, K. A. Harras, and E. Zegura (2015). “Femto clouds: leveraging mobile devices to provide cloud service at the edge”. In: *2015 IEEE 8th International Conference on Cloud Computing.* IEEE, pp. 9–16.

Hägglund, T. (2021). *AUTOMATIC CONTROL Lecture Notes.* Lund University, Lund, Sweden, p. 56.

Hansson, G. (2021). *Computation offloading of 5g devices at the edge using webassembly.*

Hu, W., Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan (2016). "Quantifying the impact of edge computing on mobile applications". In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '16. Association for Computing Machinery, Hong Kong, Hong Kong. DOI: 10.1145/2967360.2967369.

IBM (2023). *Ibm containers.* Accessed: 2023-12-15. URL: https://www.ibm.com/topics/containers.

Inseego (2023). *What is URLLC?* https://inseego.com/resources/5g-glossary/what-is-urllc/. Accessed on February 2, 2024.

Kubernetes (2023a). *Kubernetes deployment.* https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Last modified September 06, 2023 at 10:16.

Kubernetes (2023b). *Kubernetes documentation - concepts overview.* URL: https://kubernetes.io/docs/concepts/overview/.

Kubernetes (2023c). *Kubernetes github repository.* Accessed: 2023-12-18. URL: https://github.com/kubernetes/kubernetes.

Kubernetes (2023d). *Kubernetes service.* https://kubernetes.io/docs/concepts/services-networking/service/. Last modified December 08, 2023 at 6:27 PM PST.

Kubernetes (2024). *Command line tool (kubectl).* https://kubernetes.io/docs/reference/kubectl/. Accessed on 2024-02-13.

Kumar, K., J. Liu, Y.-H. Lu, and B. Bhargava (2012). "A survey of computation offloading for mobile systems". *Mobile networks and Applications* **18**, pp. 129–140.

Kwakernaak, H. and R. Sivan (1972). *Linear Optimal Control Systems.* Chap. 6.4.

Li, B., W. Dong, and Y. Gao (2021). "Wiprog: a webassembly-based approach to integrated iot programming". In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, pp. 1–10.

Luo, C., J. Nightingale, E. Asemota, and C. Grecos (2015). "A uav-cloud system for disaster sensing applications". In: *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. IEEE, pp. 1–5.

MathWorks (2023). *Dlqr function documentation.* Accessed:2023-12-28. URL: https://se.mathworks.com/help/control/ref/dlqr.html.

Mattingley, J. and S. Boyd (2012). "Cvxgen: a code generator for embedded convex optimization". *Optimization and Engineering* **13**:1, pp. 1–27.

MDN (2023). *Webassembly.* https://developer.mozilla.org/en-US/docs/WebAssembly. Accessed: 11 17, 2023.

Nurul-Hoque, M. and K. A. Harras (2021). "Nomad: cross-platform computational offloading and migration in femtoclouds using webassembly". In: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 168–178.

Park, P., S. C. Ergen, C. Fischione, C. Lu, and K. H. Johansson (2017). "Wireless network design for control systems: a survey". *IEEE Communications Surveys & Tutorials* **20**:2, pp. 978–1013.

Peng, H. (2023). *Tamin Cloud Integrated Systems in the Wild*. PhD thesis. Department of Electrical and Information Technology, Lund University.

Pigot, H. (2021). *Furuta Pendulum 2019 README*. `https://gitlab.control.lth.se/processes/furutapendulum2019/-/blob/master/README.md`. Accessed on 2024-02-10.

Rawlings, J. B., D. Q. Mayne, and M. M. Diehl (2017). *Model Predictive Control: Theory, Computation, and Design*. 2nd. Nob Hill Publishing.

Schultz, W. C. and V. C. Rideout (1961). "Control system performance measures: past, present, and future". *IRE Transactions on Automatic Control* **AC-6**:1, pp. 22–35. DOI: `10.1109/TAC.1961.6429306`.

Skarin, P. (2021). *Control over the Cloud: Offloading, Elastic Computing, and Predictive Control*. PhD thesis. Department of Automatic Control, Lund University.

Umsonst, D. and F. S. Barbosa (2024). "Remote tube-based mpc for tracking over lossy networks". In: *CDC2024 Conference*. IEEE.

WasmEdge (2023a). *WasmEdge Documentation*. Accessed on 2024-03-26. URL: `https://wasmedge.org/docs/develop/deploy/intro` (visited on 2024-03-26).

WasmEdge (2023b). *Wasmedge documentation the aot compiler*. Accessed: 2023-12-15. URL: `https://wasmedge.org/docs/start/build-and-run/aot/`.

WasmEdge Documentation Overview (2023). Accessed: 2023-12-15. URL: `https://wasmedge.org/docs/start/overview`.

WebAssembly (2023). *Overview*. `https://webassembly.org/`. Accessed: 02 15, 2024.

WebAssembly Community Group (2023). *WebAssembly Specification Release 2.0 (Draft 2023-11-07)*. Ed. by A. Rossberg. `https://webassembly.github.io/spec/`.

A

Implementation

In this appendix code developed in the thesis project is presented. Hence the CVXGEN solver code is not presented here.

A.1 Host

main.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include "util.h"

// Shared control quality cost variable
volatile double cost = 0;
// Shared flag for the file sent
volatile int fileSent = 0;

// Mutex for synchronization of the control quality variable
pthread_mutex_t cost_mutex = PTHREAD_MUTEX_INITIALIZER;
// Mutex for synchronization for util.c
pthread_mutex_t util_mutex = PTHREAD_MUTEX_INITIALIZER;

// Function declarations
extern void *control_quality_function(void *);
extern void *regul_function(void *);
extern void *sender_function(void *);

int main()
{
    // Create thread handles
    pthread_t control_quality_thread, regul_thread, sender_thread
    ;

    // Initialize util
    int status = init_util();

```

```
29     if (status)
30     {
31         // Create threads
32         pthread_create(&control_quality_thread, NULL,
       control_quality_function, NULL);
33         pthread_create(&regul_thread, NULL, regul_function, NULL)
       ;
34         pthread_create(&sender_thread, NULL, sender_function,
       NULL);
35
36         // Wait for threads to finish (this will never happen in
       this example as threads run indefinitely)
37         pthread_join(control_quality_thread, NULL);
38         pthread_join(regul_thread, NULL);
39         pthread_join(sender_thread, NULL);
40     }
41     else
42     {
43         printf("Some I/O problems\n");
44     }
45
46     // Finalize util
47     fin_util();
48
49     return 0;
50 }
```

util.c

```
1 #include "util.h"
2
3 extern pthread_mutex_t util_mutex;
4
5 // Pointer to a structure representing a Moberg device
6 struct moberg *moberg;
7
8 // Structure representing an analog output channel
9 struct moberg_analog_out analog_out_u;
10
11 // Structure representing analog input channels for various
     parameters
12 struct moberg_analog_in analog_in_phi;
13 struct moberg_analog_in analog_in_theta;
14 struct moberg_analog_in analog_in_dtPhi;
15 struct moberg_analog_in analog_in_dtTheta;
16
17 // Variables to store current and previous values of 'phi' and a
     count of rotations
18 double newPhi;
19 double oldPhi;
20 int rotationPhi = 0;
21
22 // Function to calculate the sum of two timespec structs
23 void timespec_add(struct timespec *a, struct timespec *b)
24 {
25   // Add seconds and nanoseconds of 'b' to 'a'
26   a->tv_sec += b->tv_sec;
27   a->tv_nsec += b->tv_nsec;
28
29   // Ensure nanoseconds are less than one billion
30   if (a->tv_nsec >= 1000000000UL)
31   {
32     a->tv_sec++;                // Increment seconds by 1
33     a->tv_nsec %= 1000000000UL; // Reduce nanoseconds to less
       than one billion
34   }
35 }
36
37 // Function to sleep until a specific time point
38 void sleep_until_next_activation(struct periodic_task *tsk)
39 {
40   int err;
41   // Keep sleeping until the specified time point ('tsk->
       current_activation') is reached
42   do
43   {
44     // Perform an absolute sleep until 'tsk->current_activation'
45     err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &tsk->
       current_activation, NULL);
46     // If the function wakes up due to a signal interruption,
       continue sleeping
```

```
47   } while (err != 0 && errno == EINTR); // Repeat until no error
       and errno is EINTR
48   // Ensure no error occurred during sleep
49   assert(err == 0);
50 }
51
52 // Function to initialize communication channels with a Moberg
       device
53 int init_util()
54 {
55   // Allocate memory for a new Moberg device and assign it to '
       moberg'
56   moberg = moberg_new();
57
58   // Open analog output and input channels and check for
       successful initialization
59   int status = moberg_OK(moberg_analog_out_open(moberg, 41, &
       analog_out_u));
60   status &= moberg_OK(moberg_analog_in_open(moberg, 40, &
       analog_in_phi));
61   status &= moberg_OK(moberg_analog_in_open(moberg, 41, &
       analog_in_theta));
62   status &= moberg_OK(moberg_analog_in_open(moberg, 42, &
       analog_in_dtPhi));
63   status &= moberg_OK(moberg_analog_in_open(moberg, 43, &
       analog_in_dtTheta));
64
65   // If initialization failed, return status indicating failure
66   if (!status)
67   {
68     return status;
69   }
70
71   // Read initial value of 'phi' and store it in 'newPhi', also
       set 'oldPhi' to the same value
72   analog_in_phi.read(analog_in_phi.context, &newPhi);
73   oldPhi = newPhi;
74
75   // Return status indicating successful initialization
76   return status;
77 }
78
79 // Function to read process states with signal manipulation to
       ensure desired form
80 void read_util(struct stateValues *state)
81 {
82   // Lock the mutex to ensure thread safety during reading
83   pthread_mutex_lock(&util_mutex);
84
85   // Read 'theta' value from analog input and adjust it by
       subtracting PI
86   analog_in_theta.read(analog_in_theta.context, &state->theta);
87   state->theta = state->theta - M_PI;
88
```

```
    // Read 'dtTheta' value from analog input
    analog_in_dtTheta.read(analog_in_dtTheta.context, &state->dtTheta);

    // Read 'phi' value from analog input and handle rotation counting
    analog_in_phi.read(analog_in_phi.context, &newPhi);
    if ((0 <= newPhi && newPhi <= 1) && (5.2 <= oldPhi && oldPhi < 2 * M_PI))
      rotationPhi++;
    else if ((0 <= oldPhi && oldPhi <= 1) && (5.2 <= newPhi && newPhi < 2 * M_PI))
      rotationPhi--;
    state->phi = newPhi + rotationPhi * 2 * M_PI;
    oldPhi = newPhi; // Update 'oldPhi'

    // Read 'dtPhi' value from analog input
    analog_in_dtPhi.read(analog_in_dtPhi.context, &state->dtPhi);

    // Unlock the mutex after finishing reading
    pthread_mutex_unlock(&util_mutex);
}

// Function to actuate by sending a control signal to the Moberg device
void actuate_util(double u_c)
{
    // Lock the mutex to ensure thread safety during actuation
    pthread_mutex_lock(&util_mutex);

    // Write the control signal 'u_c' to the analog output channel
    analog_out_u.write(analog_out_u.context, u_c, NULL);

    // Unlock the mutex after actuation
    pthread_mutex_unlock(&util_mutex);
}

// Function to finalize and clean up resources used by the utility
void fin_util()
{
    // Close all analog output and input channels and check for successful closure
    int status = moberg_OK(moberg_analog_out_close(moberg, 41, analog_out_u));
    status &= moberg_OK(moberg_analog_in_close(moberg, 40, analog_in_phi));
    status &= moberg_OK(moberg_analog_in_close(moberg, 41, analog_in_theta));
    status &= moberg_OK(moberg_analog_in_close(moberg, 42, analog_in_dtPhi));
    status &= moberg_OK(moberg_analog_in_close(moberg, 43, analog_in_dtTheta));

```

```
131   // If closure failed, print an error message and exit with
        failure
132   if (!status)
133   {
134     printf("%s: Can not close I/O ports\n", __FILE__);
135     exit(EXIT_FAILURE);
136   }
137
138   // Free the memory allocated for the Moberg device
139   moberg_free(moberg);
140 }
```

quality.c

```c
#include "util.h"

// Define the period for quality control in nanoseconds
#define PERIOD_IN_NANOS_QUALITY (1UL * 1000000UL)

// Shared variable to store cost
extern volatile double cost;

// Mutex for synchronization
extern pthread_mutex_t cost_mutex;

// Function responsible for controlling quality
void *control_quality_function(void *arg)
{
  // Structure to hold the state values
  struct stateValues state;

  // Variable to handle errors
  int err;

  // Structure to define periodic task properties
  struct periodic_task tsk;

  // Set the initial job ID to 1 to match real-time theory
  tsk.current_job_id = 1;

  // Indicate that the task has not terminated yet
  tsk.terminated = 0;

  // Set the desired period for the task
  tsk.period.tv_sec = 0;
  tsk.period.tv_nsec = PERIOD_IN_NANOS_QUALITY;

  // Record the time of the first job
  err = clock_gettime(CLOCK_MONOTONIC, &tsk.first_activation);
  assert(err == 0); // Ensure clock_gettime operation succeeds

  // Set the current activation time to the first activation time
  tsk.current_activation = tsk.first_activation;

  // Execute the task until termination is signaled
  while (!tsk.terminated)
  {
    // Wait until the next activation time
    sleep_until_next_activation(&tsk);

    // Read the state values using utility function
    read_util(&state);

    // Lock the mutex for accessing the shared 'cost' variable
    pthread_mutex_lock(&cost_mutex);

```

```
53      // Update the cost using a formula based on state values
54      cost = 0.5 * (state.theta * state.theta + state.dtTheta *
        state.dtTheta + state.phi * state.phi + state.dtPhi * state.
        dtPhi) + 0.5 * cost;
55
56      // Unlock the mutex after updating 'cost'
57      pthread_mutex_unlock(&cost_mutex);
58
59      // Increment the job ID for the next iteration
60      tsk.current_job_id++;
61
62      // Update the current activation time for the next iteration
63      timespec_add(&tsk.current_activation, &tsk.period);
64    }
65
66    // Task execution complete, return NULL
67    return NULL;
68  }
```

sender.c

```
#include "controller.h"

// Global variable to store the TCP socket for remote
    communication
int tcpSocketRemote;

// Structure representing the server address for TCP
    communication
struct sockaddr_in serverAddressRemoteTCP;

// Global variable to keep track of the number of times files
    have been sent
int count = 0;

// External declaration of a volatile integer variable 'fileSent'
    and a mutex 'file_mutex'
extern volatile int fileSent;
extern pthread_mutex_t file_mutex;

// Function to send data over TCP
void sendTCPData(int tcpSocket, void *data, size_t dataSize)
{
  ssize_t bytes_sent = 0;
  ssize_t remaining_bytes = dataSize;

  int c = 0;
  // Loop until all data is sent
  while (remaining_bytes > 0)
  {
    // Send data and handle errors
    bytes_sent = send(tcpSocket, data + (dataSize -
    remaining_bytes), remaining_bytes, 0);
    if (bytes_sent == -1)
    {
      perror("Error sending data");
      c++;
      // If sending fails multiple times, exit with failure
      if (c >= 5)
      {
        printf("%s: Could not send the packet.\n", __FILE__);
        exit(EXIT_FAILURE);
      }
    }
    else
    {
      remaining_bytes -= bytes_sent;
    }
  }
}

// Function to send a file over TCP
void send_file()
```

```
48 {
49   struct stat file_stat;
50   // Get file statistics
51   if (stat(FILENAME, &file_stat) == -1)
52   {
53     printf("%s: Could not get the file stats.\n", __FILE__);
54     exit(EXIT_FAILURE);
55   }
56 
57   // Extract file size from file statistics
58   off_t file_size = file_stat.st_size;
59 
60   // Convert file size to a string
61   char file_size_str[20];
62   snprintf(file_size_str, sizeof(file_size_str), "%lld", (long
       long)file_size);
63 
64   // Send file size and count over TCP
65   sendTCPData(tcpSocketRemote, file_size_str, sizeof(
       file_size_str));
66   sendTCPData(tcpSocketRemote, &count, sizeof(count));
67 
68   // Send the file contents over TCP
69   FILE *file_ptr = fopen(FILENAME, "rb");
70   if (file_ptr == NULL)
71   {
72     perror("Error opening file");
73     exit(EXIT_FAILURE);
74   }
75 
76   char buffer[4096];
77   ssize_t bytes_read;
78   while (1)
79   {
80     // Read from file and send data until the end of file is
       reached
81     bytes_read = fread(buffer, 1, sizeof(buffer), file_ptr);
82     if (bytes_read <= 0)
83     {
84       break;
85     }
86     sendTCPData(tcpSocketRemote, buffer, bytes_read);
87   }
88 
89   fclose(file_ptr);
90 
91   // Receive acknowledgment for file sent
92   int sent;
93   recv(tcpSocketRemote, &sent, sizeof(int), 0);
94   if (sent == count)
95   {
96     // printf("The file was sent=%d\n", sent);
97   }
98   else
```

```
99    {
100     printf("%s: The file id mismatch, expect: %d got: %d\n",
        __FILE__, count, sent);
101   }
102   count++;
103 }
104
105 // Function to set the IP address and port for a TCP socket
106 void setIpAddressTCP()
107 {
108   // Create a TCP socket
109   tcpSocketRemote = socket(AF_INET, SOCK_STREAM, 0);
110   if (tcpSocketRemote == -1)
111   {
112     printf("%s: Could not create a TCP socket for sending the
        file.\n", __FILE__);
113     exit(EXIT_FAILURE);
114   }
115
116   // Set TCP_NODELAY flag
117   int flag = 1;
118   int result = setsockopt(tcpSocketRemote, IPPROTO_TCP,
        TCP_NODELAY, (char *)&flag, sizeof(int));
119   if (result < 0)
120   {
121     printf("%s: Could not set no delay flag on the tcp socket.\n"
        , __FILE__);
122   }
123
124   // Configure the server address
125   memset(&serverAddressRemoteTCP, 0, sizeof(
        serverAddressRemoteTCP));
126   serverAddressRemoteTCP.sin_family = AF_INET;
127   serverAddressRemoteTCP.sin_port = htons(TCP_PORT);
128
129   // Convert IP address from text to binary form and assign to
        the server address structure
130   if (inet_pton(AF_INET, REMOTE_IP_ADDRESS, &
        serverAddressRemoteTCP.sin_addr) <= 0)
131   {
132     printf("%s: Could not convert the ip address to a network
        address.\n", __FILE__);
133     close(tcpSocketRemote);
134     exit(EXIT_FAILURE);
135   }
136
137   // Connect the TCP socket to the server address
138   if (connect(tcpSocketRemote, (struct sockaddr *)&
        serverAddressRemoteTCP, sizeof(serverAddressRemoteTCP)) ==
        -1)
139   {
140     printf("%s: Could not connect the socket to the network
        address.\n", __FILE__);
141     close(tcpSocketRemote);
```

```
142       exit(EXIT_FAILURE);
143     }
144 }
145 
146 // Function to set model parameters and send them over TCP
147 void setModelParams()
148 {
149   // Structure containing model parameters
150   struct ModelParams model = {
151       .Q0 = 250, .Q1 = 10, .Q2 = 240, .Q3 = 90, .R0 = 100, .A0 =
        1.00328136052027794, .A4 = 0.00800821520779518, .A8 = 0.0, .
        A12 = 0.0, .A1 = 0.92094590856619241, .A5 =
        1.00368177128066782, .A9 = 0, .A13 = 0, .A2 =
        -0.00019332764892868, .A6 = -0.00000051544830440, .A10 =
        0.99960007998933442, .A14 = 0.00799840021331200, .A3 =
        -0.04836477573415137, .A7 = -0.00019335342134390, .A11 = 0, .
        A15 = 1, .B0 = -0.16722232812227830, .B1 =
        -41.83679291328630256, .B2 = 0.11314832814927638, .B3 =
        28.29220011726365414, .u_max = .2, .S = 1};
152 
153   // Send model parameters over TCP
154   sendTCPData(tcpSocketRemote, &model, sizeof(struct ModelParams)
        );
155 
156   // Shutdown and close the TCP socket
157   shutdown(tcpSocketRemote, 0);
158   close(tcpSocketRemote);
159 }
160 
161 // Function for the sender thread
162 void *sender_function(void *arg)
163 {
164   // Set a delay before sending data
165   struct timespec req, rem;
166   req.tv_sec = 3;
167   req.tv_nsec = 0;
168 
169   // Sleep for the specified time
170   if (clock_nanosleep(CLOCK_MONOTONIC, 0, &req, &rem) != 0)
171   {
172     printf("%s: Sleep interrupted: %ld seconds %ld nanoseconds
        remaining\n", __FILE__, rem.tv_sec, rem.tv_nsec);
173   }
174   else
175   {
176     printf("%s: Slept for 3 second\n", __FILE__);
177   }
178 
179   // Set up TCP connection and send file and model parameters
180   setIpAddressTCP();
181   send_file();
182   setModelParams();
183   printf("%s: the file and the parameters were sent \n", __FILE__
        );
```

```
184
185   // Set the 'fileSent' flag to indicate file transmission
        completion
186   pthread_mutex_lock(&file_mutex);
187   fileSent = 1;
188   pthread_mutex_unlock(&file_mutex);
189
190   // Initialize the controller
191   init();
192
193   // Return NULL to end the sender thread
194   return NULL;
195 }
```

regul.c

```
#include "controller.h"

// Function for the regulation thread
void *regul_function(void *arg)
{
    // Structure to hold the state values
    struct stateValues state;

    // Control signal
    double u_c = 0;

    // Finalize any pending tasks
    fin();

    // Variable to handle errors
    int err;

    // Structure to define periodic task properties
    struct periodic_task tsk;

    // Set the initial job ID to 1 to match real-time theory
    tsk.current_job_id = 1;

    // Indicate that the task has not terminated yet
    tsk.terminated = 0;

    // Set the desired period for the task
    tsk.period.tv_sec = 0;
    tsk.period.tv_nsec = PERIOD_IN_NANOS;

    // Record the time of the first job
    err = clock_gettime(CLOCK_MONOTONIC, &tsk.first_activation);
    assert(err == 0); // Ensure clock_gettime operation succeeds

    // Set the current activation time to the first activation
    time
    tsk.current_activation = tsk.first_activation;

    // Execute the task until termination is signaled
    while (!tsk.terminated)
    {
        // Wait until the next activation time
        sleep_until_next_activation(&tsk);

        // Actuate with the current control signal
        actuate_util(u_c);

        // Read the state values using utility function
        read_util(&state);

        // Calculate the next control signal
        u_c = calcU(&state);
```

```
52
53         // Increment the job ID for the next iteration
54         tsk.current_job_id++;
55
56         // Update the current activation time for the next
    iteration
57         timespec_add(&tsk.current_activation, &tsk.period);
58     }
59
60     // Task execution complete, return NULL
61     return NULL;
62 }
```

controller.c

```
1 #include "controller.h"
2
3 // Conditional compilation for collecting data
4 #ifdef collectData
5 // File pointer for CSV file
6 FILE *csvFile;
7 #endif
8
9 // Global variables for cost, fileSent flag, and mutexes for
     synchronization
10 extern volatile double cost;
11 extern volatile int fileSent;
12 extern pthread_mutex_t cost_mutex;
13 extern pthread_mutex_t file_mutex;
14
15 // Global variables for UDP communication
16 int udpSocketRemote;
17 struct sockaddr_in serverAddressUDP;
18 int counter = 0;
19
20 // Function to set the IP address and port for a UDP socket
21 void setIpAddressUDP(int *udpSocket, struct sockaddr_in *
     serverAddress, char *ipAddress, int port)
22 {
23     // Create a UDP socket
24     *udpSocket = socket(AF_INET, SOCK_DGRAM, 0);
25     if (*udpSocket == -1)
26     {
27         perror("socket");
28     }
29
30     // Configure the server address
31     memset(serverAddress, 0, sizeof(*serverAddress));
32     serverAddress->sin_family = AF_INET;
33     serverAddress->sin_port = htons(port);
34
35     // Convert IP address from text to binary form and assign to
       the server address structure
36     if (inet_pton(AF_INET, ipAddress, &serverAddress->sin_addr)
       <= 0)
37     {
38         perror("inet_pton");
39     }
40 }
41
42 // Function to send data via UDP
43 void sendUDPData(int udpSocket, struct sockaddr_in serverAddress,
      void *data, size_t dataSize)
44 {
45     ssize_t sentBytes;
46     sentBytes = sendto(udpSocket, data, dataSize, 0,
47                        (struct sockaddr *)&serverAddress, sizeof(
```

```
        serverAddress));
48      if (sentBytes == -1)
49      {
50          perror("sendto");
51      }
52  }
53
54  // Function to receive data on a socket
55  int receiveData(int socket, void *receivedData, size_t dataSize)
56  {
57      ssize_t recvBytes = recv(socket, receivedData, dataSize, 0);
58      if (recvBytes == -1)
59      {
60          if (errno == EAGAIN || errno == EWOULDBLOCK)
61          {
62              return 1; // No data available
63          }
64          else
65          {
66              perror("recv");
67              return -1; // Error in receiving data
68          }
69      }
70      return 0; // Data received successfully
71  }
72
73  // Function for initialization tasks
74  void init()
75  {
76      // Create a UDP socket
77      int udpSocketRemote;
78      udpSocketRemote = socket(AF_INET, SOCK_DGRAM, 0);
79      if (udpSocketRemote == -1)
80      {
81          perror("Could not create a UDP socket");
82          exit(EXIT_FAILURE);
83      }
84
85      // Configure the server address for UDP communication
86      memset(&serverAddressUDP, 0, sizeof(serverAddressUDP));
87      serverAddressUDP.sin_family = AF_INET;
88      serverAddressUDP.sin_port = htons(UDP_PORT);
89
90      // Convert remote IP address from text to binary form and
        assign to server address structure
91      if (inet_pton(AF_INET, REMOTE_IP_ADDRESS, &serverAddressUDP.
        sin_addr) <= 0)
92      {
93          perror("Could not create the server address");
94          exit(EXIT_FAILURE);
95      }
96
97      // Set timeout for the UDP socket
98      struct timeval tv;
```

```
 99      tv.tv_sec = 0;
100      tv.tv_usec = TIMEOUT_IN_MICRO;
101      if (setsockopt(udpSocketRemote, SOL_SOCKET, SO_RCVTIMEO, (
         const void *)&tv, sizeof(tv)) == -1)
102      {
103          printf("Error in setting the socket option\n");
104      }
105  }
106
107  // Function to calculate control signal 'u' based on state values
108  double calcU(struct stateValues *state)
109  {
110  #ifdef collectData
111      // Record start time for data collection
112      struct timespec start_time, end_time;
113      clock_gettime(CLOCK_MONOTONIC, &start_time);
114  #endif
115
116      // Lock mutex for accessing 'cost' variable
117      pthread_mutex_lock(&cost_mutex);
118      double current_cost = cost;
119      pthread_mutex_unlock(&cost_mutex);
120
121      // Lock mutex for accessing 'fileSent' flag
122      pthread_mutex_lock(&file_mutex);
123      int localFlag = fileSent;
124      pthread_mutex_unlock(&file_mutex);
125
126      // Structure to hold control response
127      struct clientResponse u;
128
129      // Assign counter value to state
130      state->counter = counter;
131      counter += 1;
132
133      // Check conditions for control source
134      if (current_cost < 50 && localFlag)
135      {
136          struct clientResponse u1;
137          int byte;
138          do
139          {
140              byte = recv(udpSocketRemote, &u1, sizeof(u1),
         MSG_DONTWAIT);
141          } while (byte != -1);
142
143          // Send state data via UDP
144          sendUDPData(udpSocketRemote, serverAddressUDP, state,
         sizeof(*state));
145
146          // Receive control response from server
147          if (receiveData(udpSocketRemote, &u, sizeof(u)) == 0)
148          {
149              if (u.counter == (counter - 1))
```

```
150             {
151 #ifdef collectData
152                 // Record end time for data collection
153                 clock_gettime(CLOCK_MONOTONIC, &end_time);
154                 // Calculate elapsed time
155                 long long int elapsed_time_ns = (end_time.tv_sec
    - start_time.tv_sec) * 1000000000LL + (end_time.tv_nsec -
    start_time.tv_nsec);
156                 double elapsed_time_ms = (double)elapsed_time_ns
    / 1000000;
157                 // Record data in CSV file
158                 fprintf(csvFile, "%.6f,%.6f,%.6f,%.6f,%.6f,Remote
    , %.6f, %.6f, %d ?= %d \n", state->theta, state->dtTheta,
    state->phi, state->dtPhi, u.u_c, current_cost,
    elapsed_time_ms, counter - 1, u.counter);
159                 // Check if counter exceeds limit for data
    collection
160                 if (counter > 10000)
161                 {
162                     fclose(csvFile);
163                     exit(0);
164                 }
165 #endif
166                 return u.u_c; // Return control signal
167             }
168         }
169     }
170
171     // Calculate control signal using LQR
172     u.u_c = LQR(state);
173
174 #ifdef collectData
175     // Record data in CSV file for LQR control
176     fprintf(csvFile, "%.6f,%.6f,%.6f,%.6f,%.6f,LQR, %.6f,NAN, NAN
     ?= NAN\n", state->theta, state->dtTheta, state->phi, state->
    dtPhi, u.u_c, current_cost);
177     // Check if counter exceeds limit for data collection
178     if (counter > 10000)
179     {
180         fclose(csvFile);
181         exit(0);
182     }
183 #endif
184
185     return u.u_c; // Return control signal
186 }
187
188 // Function for LQR control calculation
189 double LQR(struct stateValues *state)
190 {
191     // Check if theta is within a specific range
192     if (state->theta <= 0.75 && state->theta >= -0.75)
193     {
194         // LQR control equation parameters
```

```
195         double a = -0.3314;
196         double b = -0.0333;
197         double c = -0.0104;
198         double d = -0.0210;
199         // Calculate control signal
200         return -10 * (a * state->theta + b * state->dtTheta + c *
     state->phi + d * state->dtPhi);
201     }
202     return 0; // Return 0 if theta is not within the specified
    range
203 }
204 
205 // Function to perform finalization tasks
206 void fin()
207 {
208 #ifdef collectData
209     // Open CSV file for appending data
210     csvFile = fopen(DATAFILENAME, "a");
211     if (csvFile == NULL)
212     {
213         printf("Error opening CSV file.\n");
214     }
215     // Write header for CSV file
216     fprintf(csvFile, "Theta,dtTheta,phi,dtPhi,u,Source,
    current_cost, elapsed_time_ms\n");
217     // Print message indicating file creation
218     printf("%s: File created \n", __FILE__);
219 #endif
220     // Close UDP socket
221     close(udpSocketRemote);
222 }
```

util.h

```
1 #ifndef UTILS_H
2 #define UTILS_H
3
4 // Include necessary libraries
5 #include <stdlib.h>
6 #include <time.h>
7 #include <assert.h>
8 #include <errno.h>
9 #include <moberg.h>
10 #include <math.h>
11 #include <pthread.h>
12 #include <stdio.h>
13 #include <unistd.h>
14 #include <string.h>
15 #include <stdbool.h>
16 #include <sys/time.h>
17 #include <arpa/inet.h>
18 #include <fcntl.h>
19 #include <sys/stat.h>
20 #include <netinet/in.h>
21 #include <sys/socket.h>
22 #include <netinet/tcp.h>
23
24 // Define constants
25 #define FILENAME "../Wasm/solver_aot.wasm" // Path to the file
26 #define collectData                        // Define for data
       collection
27
28 #define TIMEOUT_IN_MICRO 14000             // Timeout value in
       microseconds
29 #define PERIOD_IN_NANOS (15UL * 1000000UL) // Period in
       nanoseconds
30
31 #ifdef collectData
32 #define DATAFILENAME "5GData.csv" // Name of the data file
33 #endif
34
35 // Define structures
36 struct periodic_task
37 {
38   unsigned long current_job_id;
39   struct timespec period;
40   struct timespec first_activation;
41   struct timespec current_activation;
42   int terminated;
43 };
44
45 struct ModelParams
46 {
47   double Q0;
48   double Q1;
49   double Q2;
```

```
50   double Q3;
51   double R0;
52   double A0;
53   double A1;
54   double A2;
55   double A3;
56   double A4;
57   double A5;
58   double A6;
59   double A7;
60   double A8;
61   double A9;
62   double A10;
63   double A11;
64   double A12;
65   double A13;
66   double A14;
67   double A15;
68   double B0;
69   double B1;
70   double B2;
71   double B3;
72   double u_max;
73   double S;
74 };
75 
76 struct stateValues
77 {
78   double phi;
79   double theta;
80   double dtPhi;
81   double dtTheta;
82   int counter;
83 };
84 
85 // Function prototypes
86 void timespec_add(struct timespec *a, struct timespec *b);   //
       Add timespec values
87 void sleep_until_next_activation(struct periodic_task *tsk); //
       Sleep until the next activation
88 int init_util();                                             //
       Initialize utility functions
89 void read_util(struct stateValues *state);                   //
       Read utility state values
90 void actuate_util(double u_c);                               //
       Actuate utility with control signal
91 void fin_util();                                             //
       Finalize utility functions
92 
93 #endif // UTILS_H
```

controller.h

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

// Include necessary header files
#include "remote.h"

// Define a structure to represent client response
struct clientResponse
{
  double u_c;  // Control signal
  int counter; // Counter value
};

// Function prototypes
double calcU(struct stateValues *state); // Calculate control
    signal
double LQR(struct stateValues *state);   // Perform LQR control
void init();                             // Initialize controller
void fin();                              // Finalize controller

#endif // CONTROLLER_H
```

remote.h

```
#ifndef REMOTE_H
#define REMOTE_H

// Edge
#define REMOTE_IP_ADDRESS "130.235.202.230"

// Heron-01
// #define REMOTE_IP_ADDRESS "130.235.83.87"

// Heron-02
// #define REMOTE_IP_ADDRESS "130.235.83.88"

// Local
// #define REMOTE_IP_ADDRESS "127.0.0.1"

#define TCP_PORT 32000
#define UDP_PORT 31450
// #define TCP_PORT 12348
// #define UDP_PORT 12346

#endif
```

A.2 Remote Solver

main.c

```
1 #include "solver.h" // Including header file for solver functions
2
3 // External declaration of functions and thread IDs
4 extern void *receiver_thread(void *arg);
5 pthread_t receiver_thread_id;
6
7 extern void *solver_thread(void *arg);
8 pthread_t solver_thread_id;
9
10 pthread_t intermediary_thread_id;
11
12 // File descriptors for Unix and IP sockets
13 int server_fd_unix;
14 int client_fd_unix;
15 int server_fd_ip;
16
17 // Flag for Multi-Process Communication (MPC) and its mutex
18 volatile int MPC = 0;
19 pthread_mutex_t MPCmutex = PTHREAD_MUTEX_INITIALIZER;
20
21 // Array to hold parameters for loading into the WebAssembly
      module
22 WasmEdge_Value ParamsLoadBridge[27];
23
24 // WebAssembly context and async object
25 WasmEdge_VMContext *VMCxt;
26 WasmEdge_Async *Async;
27
28 // String representing a function in the WebAssembly module
29 WasmEdge_String solveBridge;
30
31 // Intermediary thread function
32 void *intermediary_thread(void *arg)
33 {
34   struct sockaddr_in client_addr_ip;
35   socklen_t client_addr_length_ip = sizeof(client_addr_ip);
36
37   // Initialize pollfd structures for polling on sockets
38   struct pollfd fds[2];
39   fds[0].fd = client_fd_unix;
40   fds[0].events = POLLIN;
41   fds[1].fd = server_fd_ip;
42   fds[1].events = POLLIN;
43
44   while (1)
45   {
46     int ret = poll(fds, 2, -1); // Poll on both sockets
      indefinitely
47
```

```
48      if (ret < 0)
49      {
50        perror("poll failed");
51        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
52        continue;
53      }
54
55      // Check if there is data to read from the Unix socket
56      if (fds[0].revents & POLLIN)
57      {
58        // Receive data from Unix socket
59        struct clientResponse u_response;
60        if (recv(client_fd_unix, &u_response, sizeof(u_response),
        0) == -1)
61        {
62          perror("Failed to get control signal from solver.c");
63          fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
64          continue;
65        }
66
67        // Send received data to IP socket
68        if (sendto(server_fd_ip, &u_response, sizeof(u_response),
        0, (struct sockaddr *)&client_addr_ip, client_addr_length_ip)
         == -1)
69        {
70          perror("Failed to send the control signal to local device
        ");
71          fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
72          continue;
73        }
74      }
75
76      // Check if there is data to read from the IP socket
77      if (fds[1].revents & POLLIN)
78      {
79        // Receive data from IP socket
80        struct stateValues state;
81        if (recvfrom(server_fd_ip, &state, sizeof(state), 0, (
        struct sockaddr *)&client_addr_ip, &client_addr_length_ip) <
        0)
82        {
83          perror("Failed to get the state from local device");
84          fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
85          continue;
86        }
87
88        // Lock mutex to check MPC flag
89        pthread_mutex_lock(&MPCmutex);
90        int localMPCFlag = MPC;
91        pthread_mutex_unlock(&MPCmutex);
```

```
92
93       // Cancel async execution if MPC flag is set
94       if (localMPCFlag)
95       {
96         WasmEdge_AsyncCancel(Async);
97       }
98
99       // Send received state to solver.c via Unix socket
100      if (send(client_fd_unix, &state, sizeof(state), 0) == -1)
101      {
102        perror("Failed to send state to solver.c");
103        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
     errno, __LINE__);
104        continue;
105      }
106
107      // Receive and discard any remaining data in the Unix
     socket buffer
108      struct clientResponse u_response;
109      int bytes_received;
110      do
111      {
112        bytes_received = recv(client_fd_unix, &u_response, sizeof
     (u_response), MSG_DONTWAIT);
113      } while (bytes_received > 0);
114    }
115  }
116  pthread_exit(NULL);
117 }
118
119 // Function to set up Unix sockets
120 void setUpUnixSockets()
121 {
122   int UNIXsockets[2];
123   if (socketpair(AF_UNIX, SOCK_DGRAM, 0, UNIXsockets) == -1)
124   {
125     perror("%s : Unix socket creation failed");
126     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
     errno, __LINE__);
127     exit(EXIT_FAILURE);
128   }
129   server_fd_unix = UNIXsockets[0];
130   client_fd_unix = UNIXsockets[1];
131
132   printf("%s: UNIX UDP socket created \n", __FILE__);
133 }
134
135 // Function to set up IP sockets
136 void setUpIpSockets()
137 {
138   // Create a socket for IP communication
139   if ((server_fd_ip = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
140   {
141     perror("Server IP socket creation failed");
```

```
142     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
143     exit(EXIT_FAILURE);
144   }
145
146   // Set up server address structure for IP communication
147   struct sockaddr_in server_addr_ip;
148   server_addr_ip.sin_family = AF_INET;
149   server_addr_ip.sin_addr.s_addr = INADDR_ANY;
150   server_addr_ip.sin_port = htons(UDP_PORT);
151
152   // Bind server socket to the specified address
153   if (bind(server_fd_ip, (struct sockaddr *)&server_addr_ip,
        sizeof(server_addr_ip)) < 0)
154   {
155     perror("Server IP socket failed");
156     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
157     close(server_fd_ip);
158     exit(EXIT_FAILURE);
159   }
160
161   printf("%s: IP UDP socket created and bound \n", __FILE__);
162 }
163
164 int main()
165 {
166   // Create and join receiver thread
167   if (pthread_create(&receiver_thread_id, NULL, receiver_thread,
        NULL) != 0)
168   {
169     perror("Failed to create the receiver thread");
170     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
171     exit(EXIT_FAILURE);
172   }
173   pthread_join(receiver_thread_id, NULL);
174
175   // Set up Unix and IP sockets
176   setUpIpSockets();
177   setUpUnixSockets();
178
179   // Initialize WebAssembly context and configure it
180   WasmEdge_ConfigureContext *ConfCxt = WasmEdge_ConfigureCreate()
        ;
181   WasmEdge_ConfigureAddHostRegistration(ConfCxt,
        WasmEdge_HostRegistration_Wasi);
182
183   VMCxt = WasmEdge_VMCreate(ConfCxt, NULL);
184   WasmEdge_String initBridge = WasmEdge_StringCreateByCString("
        initBridge");
185   WasmEdge_String loadBridge = WasmEdge_StringCreateByCString("
        loadBridge");
186   solveBridge = WasmEdge_StringCreateByCString("solveBridge");
```

```
187
188    // Load WebAssembly file into the program
189    WasmEdge_Result Res;
190    Res = WasmEdge_VMLoadWasmFromFile(VMCxt, FILENAME);
191    if (!WasmEdge_ResultOK(Res))
192    {
193      printf("%s: Loading phase failed: %s\n", FILE,
       WasmEdge_ResultGetMessage(Res));
194      exit(EXIT_FAILURE);
195    }
196
197    // Validate the loaded WebAssembly module
198    Res = WasmEdge_VMValidate(VMCxt);
199    if (!WasmEdge_ResultOK(Res))
200    {
201      printf("%s: Validation phase failed: %s\n", FILE,
       WasmEdge_ResultGetMessage(Res));
202      exit(EXIT_FAILURE);
203    }
204
205    // Instantiate the WebAssembly module
206    Res = WasmEdge_VMInstantiate(VMCxt);
207    if (!WasmEdge_ResultOK(Res))
208    {
209      printf("%s: Instantiation phase failed: %s\n", FILE,
       WasmEdge_ResultGetMessage(Res));
210      exit(EXIT_FAILURE);
211    }
212
213    // Execute the 'initBridge' function in the WebAssembly module
214    Res = WasmEdge_VMExecute(VMCxt, initBridge, NULL, 0, NULL, 0);
215    if (!WasmEdge_ResultOK(Res))
216    {
217      printf("%s: initBridge function call failed: %s\n", FILE,
       WasmEdge_ResultGetMessage(Res));
218      exit(EXIT_FAILURE);
219    }
220
221    // Call the 'loadBridge' function in the WebAssembly module
       with parameters
222    Res = WasmEdge_VMExecute(VMCxt, loadBridge, ParamsLoadBridge,
       27, NULL, 0);
223    if (!WasmEdge_ResultOK(Res))
224    {
225      printf("%s: loadBridge function call failed: %s\n", FILE,
       WasmEdge_ResultGetMessage(Res));
226      exit(EXIT_FAILURE);
227    }
228
229    printf("%s: Starting solver and intermediary \n", FILE);
230
231    // Create solver thread
232    if (pthread_create(&solver_thread_id, NULL, solver_thread, (
       void *)&server_fd_unix) != 0)
```

```
233   {
234     perror("Failed to create solver thread");
235     fprintf(stdout, "%s: errno %d at line %d. \n", FILE, errno,
        LINE);
236     exit(EXIT_FAILURE);
237   }
238
239   // Create intermediary thread
240   if (pthread_create(&intermediary_thread_id, NULL,
        intermediary_thread, NULL) != 0)
241   {
242     perror("Failed to create intermediary thread");
243     fprintf(stdout, "%s: errno %d at line %d. \n", FILE, errno,
        LINE);
244     exit(EXIT_FAILURE);
245   }
246
247   // Cleanup: delete strings
248   WasmEdge_StringDelete(initBridge);
249   WasmEdge_StringDelete(loadBridge);
250
251   // Wait for the intermediary thread to finish
252   pthread_join(intermediary_thread_id, NULL);
253
254   return 0;
255 }
```

receiver.c

```c
#include "solver.h" // Including header file for solver functions

extern WasmEdge_Value ParamsLoadBridge[27]; // External
    declaration of ParamsLoadBridge array

// Function to receive a file over a TCP socket
void getFile(int sockfd)
{
  char file_size_str[20];
  off_t expected_file_size;
  ssize_t bytes_received;
  int c = 0;

  // Receive file size
  do
  {
    bytes_received = recv(sockfd, file_size_str, sizeof(
    file_size_str), 0);
    if (bytes_received < 0)
    {
      perror("File size packet not received");
      fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
    errno, __LINE__);
      if (c++ > 5)
      {
        close(sockfd);
        exit(EXIT_FAILURE);
      }
    }
  } while (bytes_received <= 0);

  file_size_str[bytes_received] = '\0';
  expected_file_size = atoll(file_size_str);

  // Receive file ID
  int file_id;
  c = 0;
  do
  {
    bytes_received = recv(sockfd, &file_id, sizeof(file_id), 0);
    if (bytes_received < 0)
    {
      perror("ID packet not received");
      fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
    errno, __LINE__);
      if (c++ > 5)
      {
        close(sockfd);
        exit(EXIT_FAILURE);
      }
    }
  } while (bytes_received <= 0);
```

```
49
50   printf("%s: Received file %d with size: %s\n", __FILE__,
       file_id, file_size_str);
51
52   // Open file for writing
53   FILE *file_ptr;
54   c = 0;
55   do
56   {
57     file_ptr = fopen(FILENAME, "wb");
58     if (file_ptr == NULL)
59     {
60       perror("The file could not be opened");
61       fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
       errno, __LINE__);
62       if (c++ > 5)
63       {
64         close(sockfd);
65         remove(FILENAME);
66         exit(EXIT_FAILURE);
67       }
68     }
69   } while (file_ptr == NULL);
70
71   // Receive file data
72   char buffer[BUFFER_SIZE];
73   off_t received_file_size = 0;
74
75   while (received_file_size < expected_file_size)
76   {
77     int num_of_buffs = (expected_file_size - received_file_size)
       / sizeof(buffer);
78     c = 0;
79     do
80     {
81       if (num_of_buffs > 0)
82         bytes_received = recv(sockfd, buffer, sizeof(buffer), 0);
83       else
84         bytes_received = recv(sockfd, buffer, expected_file_size
       - received_file_size, 0);
85
86       if (bytes_received < 0)
87       {
88         perror("File data packet not received");
89         fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
       errno, __LINE__);
90         if (c++ > 5)
91         {
92           close(sockfd);
93           remove(FILENAME);
94           exit(EXIT_FAILURE);
95         }
96       }
97     } while (bytes_received <= 0);
```

```
98
99     // Write received data to file
100    size_t written_bytes;
101    size_t total_written_bytes = 0;
102    c = 0;
103    do
104    {
105      written_bytes = fwrite(buffer + total_written_bytes, 1,
       bytes_received - total_written_bytes, file_ptr);
106      if (written_bytes > 0)
107      {
108        total_written_bytes += written_bytes;
109      }
110      else
111      {
112        perror("File data could not be written to the file");
113        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
       errno, __LINE__);
114        if (c++ > 5)
115        {
116          close(sockfd);
117          remove(FILENAME);
118          exit(EXIT_FAILURE);
119        }
120      }
121    } while (total_written_bytes < bytes_received);
122
123    received_file_size += bytes_received;
124  }
125
126  // Close file
127  if (fclose(file_ptr) != 0)
128  {
129    perror("File can not be closed");
130    fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
       errno, __LINE__);
131  }
132
133  // Check if file received completely and send acknowledgment
134  if (received_file_size == expected_file_size)
135  {
136    c = 0;
137    int bytes_sent;
138    do
139    {
140      bytes_sent = send(sockfd, &file_id, sizeof(file_id), 0);
141      if (bytes_sent == -1)
142      {
143        perror("File id could not be sent to the client");
144        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
       errno, __LINE__);
145        if (c++ > 5)
146        {
147          close(sockfd);
```

```
148            remove(FILENAME);
149            exit(EXIT_FAILURE);
150          }
151        }
152        else if (bytes_sent < sizeof(file_id))
153        {
154          printf("%s: Not all data was sent at line   %s\n",
        __FILE__, __LINE__);
155          close(sockfd);
156        }
157      } while (bytes_sent == -1);
158
159      printf("%s: File received and saved as %s\n", __FILE__,
        FILENAME);
160    }
161    else
162    {
163      printf("%s: File size mismatch. File not saved. The file
        received was %ld expected %ld \n", __FILE__,
        received_file_size, expected_file_size);
164      remove(FILENAME);
165      close(sockfd);
166      exit(EXIT_FAILURE);
167    }
168  }
169
170  // Function to retrieve model parameters over TCP socket
171  void retrieveModelParams(int sockfd)
172  {
173    int c = 0;
174    ssize_t bytes_received;
175    struct ModelParams params;
176
177    // Receive model parameters
178    do
179    {
180      bytes_received = recv(sockfd, &params, sizeof(struct
        ModelParams), 0);
181      if (bytes_received < 0)
182      {
183        perror("Parameter packet not received");
184        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
185        if (c++ > 5)
186        {
187          close(sockfd);
188          exit(EXIT_FAILURE);
189        }
190      }
191      else if (bytes_received != sizeof(struct ModelParams))
192      {
193        printf("%s: Parameter packet size not right", __FILE__);
194        close(sockfd);
195        exit(EXIT_FAILURE);
```

```
196     }
197   } while (bytes_received <= 0);
198 
199   // Convert received parameters to WebAssembly values and store
        in ParamsLoadBridge array
200   ParamsLoadBridge[0] = WasmEdge_ValueGenF64(params.Q0);
201   ParamsLoadBridge[1] = WasmEdge_ValueGenF64(params.Q1);
202   ParamsLoadBridge[2] = WasmEdge_ValueGenF64(params.Q2);
203   ParamsLoadBridge[3] = WasmEdge_ValueGenF64(params.Q3);
204   ParamsLoadBridge[4] = WasmEdge_ValueGenF64(params.R0);
205   ParamsLoadBridge[5] = WasmEdge_ValueGenF64(params.A0);
206   ParamsLoadBridge[6] = WasmEdge_ValueGenF64(params.A1);
207   ParamsLoadBridge[7] = WasmEdge_ValueGenF64(params.A2);
208   ParamsLoadBridge[8] = WasmEdge_ValueGenF64(params.A3);
209   ParamsLoadBridge[9] = WasmEdge_ValueGenF64(params.A4);
210   ParamsLoadBridge[10] = WasmEdge_ValueGenF64(params.A5);
211   ParamsLoadBridge[11] = WasmEdge_ValueGenF64(params.A6);
212   ParamsLoadBridge[12] = WasmEdge_ValueGenF64(params.A7);
213   ParamsLoadBridge[13] = WasmEdge_ValueGenF64(params.A8);
214   ParamsLoadBridge[14] = WasmEdge_ValueGenF64(params.A9);
215   ParamsLoadBridge[15] = WasmEdge_ValueGenF64(params.A10);
216   ParamsLoadBridge[16] = WasmEdge_ValueGenF64(params.A11);
217   ParamsLoadBridge[17] = WasmEdge_ValueGenF64(params.A12);
218   ParamsLoadBridge[18] = WasmEdge_ValueGenF64(params.A13);
219   ParamsLoadBridge[19] = WasmEdge_ValueGenF64(params.A14);
220   ParamsLoadBridge[20] = WasmEdge_ValueGenF64(params.A15);
221   ParamsLoadBridge[21] = WasmEdge_ValueGenF64(params.B0);
222   ParamsLoadBridge[22] = WasmEdge_ValueGenF64(params.B1);
223   ParamsLoadBridge[23] = WasmEdge_ValueGenF64(params.B2);
224   ParamsLoadBridge[24] = WasmEdge_ValueGenF64(params.B3);
225   ParamsLoadBridge[25] = WasmEdge_ValueGenF64(params.u_max);
226   ParamsLoadBridge[26] = WasmEdge_ValueGenF64(params.S);
227 }
228 
229 // Function to set up TCP sockets and handle file and parameter
        retrieval
230 int setupSockets()
231 {
232   // Creating a TCP socket for file and parameters
233   int server_fd = socket(AF_INET, SOCK_STREAM, 0);
234   if (server_fd == -1)
235   {
236     perror("Server TCP socket");
237     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
238     exit(EXIT_FAILURE);
239   }
240 
241   printf("%s: Server TCP socket created\n", __FILE__);
242   struct sockaddr_in tcpAddr;
243   tcpAddr.sin_family = AF_INET;
244   tcpAddr.sin_addr.s_addr = INADDR_ANY;
245   tcpAddr.sin_port = htons(TCP_PORT);
246 
```

```
247   // Binding the socket
248   if (bind(server_fd, (struct sockaddr *)&tcpAddr, sizeof(tcpAddr
        )) == -1)
249   {
250     perror("Server TCP socket not bound");
251     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
252     close(server_fd);
253     exit(EXIT_FAILURE);
254   }
255   printf("%s: Server TCP socket bound\n", __FILE__);
256
257   // Listening for connections
258   if (listen(server_fd, 5) == -1)
259   {
260     perror("Server TCP socket not listening");
261     fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
262     close(server_fd);
263     exit(EXIT_FAILURE);
264   }
265   printf("%s: Server TCP listening on port %d\n", __FILE__,
        TCP_PORT);
266
267   int client_fd;
268   int c = 0;
269   struct sockaddr_in client_addr;
270   socklen_t client_addr_len = sizeof(client_addr);
271
272   // Accepting client connection
273   do
274   {
275     client_fd = accept(server_fd, (struct sockaddr *)&client_addr
        , &client_addr_len);
276     if (client_fd == -1)
277     {
278       perror("Client TCP not accepted");
279       fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
        errno, __LINE__);
280       if (c++ > 5)
281       {
282         close(server_fd);
283         exit(EXIT_FAILURE);
284       }
285     }
286   } while (client_fd == -1);
287
288   printf("%s: Client TCP socket connected\n", __FILE__);
289
290   // Close the server socket and return client socket
291   close(server_fd);
292   return client_fd;
293 }
294
```

```
295 // Receiver thread function
296 void *receiver_thread(void *arg)
297 {
298   // Setup TCP socket and retrieve file and model parameters
299   int client_fd = setupSockets();
300   getFile(client_fd);
301   retrieveModelParams(client_fd);
302 
303   // Inform user and close the socket
304   printf("%s: File and model parameters are received closing
        socket.\n", __FILE__);
305   close(client_fd);
306 
307   return NULL;
308 }
```

solver.c

```
#include "solver.h" // Including header file for solver functions

// External declarations
extern WasmEdge_VMContext *VMCxt;
extern WasmEdge_Async *Async;
extern WasmEdge_String solveBridge;

// Array to store states
WasmEdge_Value states[4];

// External variables
extern volatile int MPC;
extern pthread_mutex_t MPCmutex;

// Solver thread function
void *solver_thread(void *arg)
{
  // Extracting server file descriptor from argument
  int server_fd = *(int *)arg;

  // Continuous loop to receive and solve states
  while (1)
  {
    struct stateValues state;
    struct clientResponse u;

    // Receiving state from server
    ssize_t bytes_received = recv(server_fd, &state, sizeof(state
    ), 0);
    if (bytes_received != sizeof(state))
    {
      perror("Failed to get the state");
      fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
    errno, __LINE__);
      continue;
    }

    // Converting received states to WebAssembly values
    states[0] = WasmEdge_ValueGenF64(state.phi);
    states[1] = WasmEdge_ValueGenF64(state.theta);
    states[2] = WasmEdge_ValueGenF64(state.dtPhi);
    states[3] = WasmEdge_ValueGenF64(state.dtTheta);
    WasmEdge_Value U[1];

    // Setting MPC flag
    pthread_mutex_lock(&MPCmutex);
    MPC = 1; // Set flag
    pthread_mutex_unlock(&MPCmutex);

    // Asynchronously executing solveBridge function with states
    Async = WasmEdge_VMAsyncExecute(VMCxt, solveBridge, states,
    4);
```

```
50      WasmEdge_Result Res = WasmEdge_AsyncGet(Async, U, 1);
51
52      // Resetting MPC flag
53      pthread_mutex_lock(&MPCmutex);
54      MPC = 0; // Reset flag
55      pthread_mutex_unlock(&MPCmutex);
56
57      // Checking if execution was successful
58      if (!WasmEdge_ResultOK(Res))
59      {
60        printf("%s: solver got interrupted error with code: %s \n",
       __FILE__, WasmEdge_ResultGetMessage(Res));
61        continue;
62      }
63
64      // Retrieving control response from solver
65      u.u_c = WasmEdge_ValueGetF64(U[0]);
66      u.counter = state.counter;
67
68      // Sending control response to server
69      ssize_t bytes_sent = send(server_fd, &u, sizeof(u), 0);
70      if (bytes_sent == -1)
71      {
72        perror("Failed to send the control signal");
73        fprintf(stdout, "%s: errno %d at line %d. \n", __FILE__,
      errno, __LINE__);
74        continue;
75      }
76
77      // Clearing any remaining data in the receive buffer
78      bytes_received;
79      do
80      {
81        bytes_received = recv(server_fd, &state, sizeof(state),
      MSG_DONTWAIT);
82      } while (bytes_received > 0);
83    }
84
85    return NULL;
86  }
```

solver.h

```
#ifndef SOLVER_H
#define SOLVER_H

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <errno.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <sys/types.h>
#include <poll.h>

#include <wasmedge/wasmedge.h>

#define FILENAME "solver_aot.wasm" // Filename of the WebAssembly
    module
#define TCP_PORT 12348              // TCP port number
#define BUFFER_SIZE 4096            // Size of the buffer for data
    transmission

#define MAX_THREADS 10 // Maximum number of threads
#define UDP_PORT 12346 // UDP port number

// Structure defining model parameters
struct ModelParams
{
  double Q0;
  double Q1;
  double Q2;
  double Q3;
  double R0;
  double A0;
  double A1;
  double A2;
  double A3;
  double A4;
  double A5;
  double A6;
  double A7;
  double A8;
  double A9;
  double A10;
  double A11;
  double A12;
  double A13;
  double A14;
  double A15;
  double B0;
  double B1;
```

```
51   double B2;
52   double B3;
53   double u_max;
54   double S;
55 };
56 
57 // Structure defining state values
58 struct stateValues
59 {
60   double phi;
61   double theta;
62   double dtPhi;
63   double dtTheta;
64   int counter;
65 };
66 
67 // Structure defining client response
68 struct clientResponse
69 {
70   double u_c;
71   int counter;
72 };
73 
74 // Structure for cleanup arguments
75 typedef struct
76 {
77   WasmEdge_VMContext *vmCxt;
78   WasmEdge_ConfigureContext *confCxt;
79   WasmEdge_String loadBridge;
80 } CleanupArgs;
81 
82 #endif // SOLVER_H
```

Containerfile

```
FROM fedora:38

# Update and install necessary packages
RUN dnf update -y && dnf install gcc python git which -y

# Install WasmEdge using the provided script
RUN curl -sSf https://raw.githubusercontent.com/WasmEdge/WasmEdge
    /master/utils/install.sh | bash

# Set the working directory
WORKDIR /testWasmEdge/

# Copy the local files to the container
COPY . /testWasmEdge/

# Expose UDP and TCP ports
EXPOSE 12346/udp
EXPOSE 12348

# Compile the solver.c file with WasmEdge libraries
RUN gcc solver.c -lwasmedge -I/root/.wasmedge/include -L/root/.
    wasmedge/lib -o solver.exe

# Set the entry point for the container
ENTRYPOINT ./solver.exe
```

solver-dep-sol.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: solver-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: solver
  template:
    metadata:
      labels:
        app: solver
    spec:
      containers:
        - name: solver-container
          image: mralbayati/solver
          ports:
            - containerPort: 12346
              name: udpport
              protocol: UDP
            - containerPort: 12348
              name: tcpport
              protocol: TCP

---
apiVersion: v1
kind: Service
metadata:
  name: solver-service
spec:
  selector:
    app: solver
  ports:
    - protocol: UDP
      name: udpport
      port: 12346
      targetPort: 12346
      nodePort: 31450
    - protocol: TCP
      name: tcpport
      port: 12348
      targetPort: 12348
      nodePort: 32000
  type: NodePort
```

A.3 AOT Wasm MPC Solver

bridge.c

```
#include "./cvxgen/solver.h"        // Include the solver header
    file
#include "bridge.h"                 // Include the bridge header
    file
#include <emscripten/emscripten.h> // Include Emscripten headers

Vars vars;          // Declaration of variables from solver
Params params;      // Declaration of parameters from solver
Workspace work;     // Declaration of workspace from solver
Settings settings; // Declaration of settings from solver

int main() { return 0; } // Entry point of the program, currently
     empty

// Function to initialize the bridge
EMSCRIPTEN_KEEPALIVE void initBridge()
{
  set_defaults();    // Set default values for solver parameters
  setup_indexing(); // Setup indexing for solver
}

// Function to load bridge parameters
EMSCRIPTEN_KEEPALIVE void loadBridge(
    double Q0, double Q1, double Q2, double Q3,
    double R0,
    double A0, double A1, double A2, double A3,
    double A4, double A5, double A6, double A7,
    double A8, double A9, double A10, double A11,
    double A12, double A13, double A14, double A15,
    double B0, double B1, double B2, double B3,
    double u_max, double S)
{
  // Load parameters for solver from the provided values
  params.Q[0] = Q0;
  params.Q[1] = Q1;
  params.Q[2] = Q2;
  params.Q[3] = Q3;

  params.R[0] = R0;

  params.A[0] = A0;
  params.A[1] = A1;
  params.A[2] = A2;
  params.A[3] = A3;
  params.A[4] = A4;
  params.A[5] = A5;
  params.A[6] = A6;
  params.A[7] = A7;
  params.A[8] = A8;
```

```
47   params.A[9] = A9;
48   params.A[10] = A10;
49   params.A[11] = A11;
50   params.A[12] = A12;
51   params.A[13] = A13;
52   params.A[14] = A14;
53   params.A[15] = A15;
54
55   params.B[0] = B0;
56   params.B[1] = B1;
57   params.B[2] = B2;
58   params.B[3] = B3;
59
60   params.u_max[0] = u_max;
61   params.S[0] = S;
62 }
63
64 // Function to solve bridge given state values and return control
       signal
65 EMSCRIPTEN_KEEPALIVE double solveBridge(double phi, double theta,
       double dtPhi, double dtTheta)
66 {
67   // Set initial state values for solver
68   params.x_0[0] = theta;
69   params.x_0[1] = dtTheta;
70   params.x_0[2] = phi;
71   params.x_0[3] = dtPhi;
72
73   settings.verbose = 0; // Set verbosity of solver settings
74   solve();              // Solve the optimization problem
75
76   // Check if the solver has converged and return appropriate
       control signal
77   if (work.converged == 1)
78   {
79     if (params.x_0[0] <= 0.75 && params.x_0[0] >= -0.75)
80     {
81       return 10 * vars.u_0[0]; // Return control signal
82     }
83   }
84
85   // Return default value if solver has not converged
86   return 0;
87 }
88
89 // Function to return control signal 2
90 EMSCRIPTEN_KEEPALIVE double returnU2()
91 {
92   // Check if the solver has converged and return appropriate
       control signal
93   if (work.converged == 1)
94   {
95     if (params.x_0[0] <= 0.75 && params.x_0[0] >= -0.75)
96     {
```

```
97          return 10 * vars.u_1[0]; // Return control signal
98        }
99    }
100
101   // Return default value if solver has not converged
102   return 0;
103 }
```

bridge.h

```
#ifndef BRIDGE_H
#define BRIDGE_H

void initBridge();

void loadBridge(
    double Q0, double Q1, double Q2, double Q3,
    double R0,
    double A0, double A1, double A2, double A3,
    double A4, double A5, double A6, double A7,
    double A8, double A9, double A10, double A11,
    double A12, double A13, double A14, double A15,
    double B0, double B1, double B2, double B3,
    double u_max, double S);

double solveBridge(double phi, double theta, double dtPhi, double
     dtTheta);

double returnU2();
#endif
```

Containerfile

```
FROM fedora:38

# Update packages and install necessary dependencies
RUN dnf update -y && dnf install gcc python git which -y

# Install WasmEdge using installation script
RUN curl -sSf https://raw.githubusercontent.com/WasmEdge/WasmEdge
    /master/utils/install.sh | bash

# Set working directory inside the container
WORKDIR /testWasmEdge/

# Copy the local files to the working directory inside the
    container
COPY . /testWasmEdge/

# Compile solver.wasm to solver_aot.wasm using WasmEdge
RUN /root/.wasmedge/bin/wasmedge compile --interruptible solver.
    wasm solver_aot.wasm
```

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
May 2024

Document Number
TFRT-6231

Author(s)
Ahmed Dhiaa Tariq Al Bayati

Supervisor
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden
Johan Eker, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Dynamic Offloading of Control Algorithms to the Edge using 5G and WebAssembly

Abstract

The aim of this work was to test if WebAssembly universal byte code and 5G communication technology is suitable in the context of offloading control mission of real-time systems. To test these tools a new dynamic offloading framework was implemented and tested on a Furuta pendulum, an inherently unstable and time-critical process using, among other computing units, an edge node as the offloading target. The implementation is considered dynamic because: 1) The local device which interacts with the I/O of the process dynamically send the control application to be used by the edge node. 2) The local device dynamically decides on which controller should control the process, either the local fallback linear quadratic regulator controller or the CVXGEN Model Predictive Control solver written in an Ahead-of-Time WebAssembly format, which is used by the edge node. The work concluded that Wasm run in interpreted form was too slow to control the process but AOT compiled Wasm which could be run by WasmEdge runtime worked well with an execution time close to the native speed. It was also concluded that data transfer using 5G technology without uRLCC was fast enough to balance the pendulum and is suitable for offloading, other communication medium where also tested in this work such as: Wi-Fi and cabled Ethernet. In the study we also developed a control quality measure for decision making on when to offload.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-113

Recipient's notes

Security classification

http://www.control.lth.se/publications/