Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design and Implementation of Synchronization Mechanisms for Distributed Pervasive Simulations

Justin Schiel

Course of Study:	B.Sc. Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Dr. rer. nat. Frank Dürr
Commenced:	July 10, 2022
Completed:	January 10, 2023

Abstract

The use of mobile devices has increased in recent years. Even though the calculation capacity of these devices is ever growing there comes a point at which a calculation is too complex for the mobile device to handle standalone. An example for this are complex mobile simulations. These simulations can be offloaded to an edge cloud server infrastructure to assist with the calculation. The vertical scalability for such edge clouds is inherently limited by the use of consumer grade hardware. Therefore horizontal scaling is needed to improve performance. The horizontal scaling implies multiple servers that are calculating the simulation asynchronously but the mobile device can only process and display the information synchronously. This creates the need for synchronizing the frame rate and output of those servers. Our proposed solution to this synchronization problem is a feedback control loop that uses an LQ-Regulator to adjust the CPU frequencies of the edge cloud servers. The controller gets the current synchronization state from the mobile device and calculates based on this regulation values that the servers use to adapt their calculation speed (by changing their CPU frequencies). To exchange messages in the distributed control system MQTT is used. To prove the validity of our approach we implemented a proof of concept. The results of an evaluation of this proof of concept show that our approach is effective for a heterogeneous workload and server setup.

Kurzfassung

Die Popularität von elektronischen mobilen Geräten hat in den letzten Jahren stark zugenommen. Obwohl die Rechenkapazität dieser mobilen Geräte immer besser wird, gibt es eine Grenze, an der Berechnungen zu komplex für das mobile Gerät werden. Ein Beispiel für einen derartigen Anwendungsfall ist eine komplexe mobile Simulation. Diese Simulation kann in eine Edge-Cloud ausgelagert werden. Da für Edge-Clouds in der Regel Endbenutzer Hardware verwendet wird, ist die vertikale Skalierbarkeit hier ebenfalls begrenzt. Daher wird horizontale Skalierung benötigt, um die Leistung zu erhöhen. Die verschiedenen Server bei einer horizontalen Skalierung berechnen Teile der Simulation asynchron, das mobile Gerät kann allerdings die Informationen nur synchron anzeigen. Daher wird ein Mechanismus benötigt, der die Frame-Rate und Ausgabe der Server synchronisiert. Unsere vorgeschlagene Lösung für dieses Problem ist ein Regelkreis mit Rückkopplung, der einen LQ-Regler verwendet, um die CPU Frequenzen der Edge-Cloud Server anzupassen. Der Regler bekommt den aktuellen synchronisations Zustand vom mobilen Gerät und berechnet basierend auf dieser Information Regelwerte für die Server, welche diese verwenden, um ihre CPU Frequenz anzupassen. Die Regelnachrichten in unserem System werden mithilfe von MQTT ausgetauscht. Um die Effektivität unseres Ansatzes zu beweisen, haben wir einen Prototyp entwickelt. Die Ergebnisse der Evaluierung dieses Prototyps zeigen, dass unser System effektiv für heterogene Rechenlast und Server Zusammenstellungen ist.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The use of AR- and VR devices has gained popularity in recent years. This technology offers a wide range of benefits to standard display technologies. The user can be immersed more in the content that he consumes, which offers a better user experience and keeps the user longer engaged in what he is doing. The immersion increases with the quality of the data displayed to the user [Boa13]. This provides a challenge for AR- and VR devices since displaying data in real time, in a sufficiently high resolution, is computationally intensive. Because the devices need to be small enough to be wearable, the power of the devices cannot be increased to an arbitrary degree. Therefore it is necessary to outsource the calculation to a more powerful device.

An application is to display a muscle simulation as an overlay over an arm in real time using an AR-Device. The muscle simulation itself is too complex to be calculated in real-time, therefore surrogate models based on neural networks have been developed to make it possible to calculate the simulation as an approximation in real time. Previous work already took care of outsourcing the neural network calculation from the AR-Device, as even the neural network approximation is too complex to be calculated on the AR-Device.

To distribute the calculation to multiple on-site computers it is necessary to split the neural network in a way that each divided neural network is calculating a part of the simulation, e.g. one neural network is calculating the upper arm and the other is calculating the under arm. This division of the neural networks has also already been researched in previous work.

What remains an open research topic is a method to synchronize the computers (we call them servers) in a way that they output the calculated data simultaneously. The purpose of this work is therefore to find a synchronization method to synchronize multiple servers for the calculation of a distributed simulation.

To this end, we propose an approach using LQ regulators in a feedback control loop to adjust the CPU frequencies of the servers such that they produce results at the same rate and same time (synchronized). This process we call LQ-CPU-Frequency-Synchronization.

Our evaluations show that using LQ-CPU-Frequency-Synchronization is effective to synchronize multiple servers, which can be heterogeneous with respect to computation speed and executing calculations of different complexity.

The remainder of this work is structured as follows:

Chapter 2 explains background knowledge to give an introduction to two main methods and technologies used by our approach, namely, LQ regulators and pub/sub communication using MQTT. We use an LQ-Regulator for controlling the synchronization and MQTT to communicate between the synchronized server. Moreover, Chapter 2 also provides an overview and discussion of related work.

Chapter 3 describes the system model, where we describe all components that are involved in the LQ-CPU-Frequency-Synchronization process together with our assumptions. This is followed by the problem statement, where the goal of this work is described.

Chapter 4 contains parts of our main contribution. To our knowledge, there is no related work that uses an LQ-Regulator to regulate the CPU speed of a server. This is why we performed a preliminary study to check whether it is feasible to do so. Followed by this preliminary study is our actual approach which explains how the LQ-CPU-Frequency-Synchronization method works in detail.

Chapter 5 gives an overview over our proof of concept implementation.

Chapter 6 shows the evaluation. We evaluated different setups and show how effective and efficient our method is working. The evaluation showed some limitations of our approach, which are discussed at the end of this chapter.

Chapter 7 summarizes this work and presents an outlook onto future work.

2 Background and Related Work

In this chapter background knowledge to the two main components (the LQ-Regulator and MQTT) is provided as well as the related work.

2.1 Background

This section contains an introduction into LQ-Regulators and MQTT to get a basic understanding over the used technologies in the preliminary study and final approach. The LQ-Regulator will be used to synchronize multiple servers with each other. To make communication between the servers and regulator possible, the pub/sub system MQTT is used.

2.1.1 LQ-Regulator

A Linear Quadratic Regulator or LQ-Regulator will be used to adjust the calculation speed of the servers that are calculating parts of the image for the AR-Device. This section will provide an overview over LQ-Regulators in order to understand the regulation concept later.

Like the well known PID-Regulator, which is a basic controller that regulates the system based on the three variations of the current state (proportional, integral, derivative), the LQ-Regulator is also a feedback controller. This means it is continuously taking the current state of the regulated system and adjusts the inputs in order to bring the system to a desired state [God03]. The basic structure of a feedback loop controller is shown in Figure 2.1. The controller takes the output of the system as input and calculates the appropriate input for the system in order to regulate it as desired. The system is then fed with the regulator output as input. In real world applications the system always behaves not exactly like predicted which is why there is a interference added in the system. Then the controller again takes the output of the system and so on.

In the case of the LQ-Regulator the state x is defined by an n dimensional vector. The regulator tries to minimize the absolute value of the state vector ($\lim_{t\to\infty} |x_t| = 0$ with x_t being the state vector at time t). System input u is defined as an m dimensional vector. The input vector values are defined by a set of linear differential equations [Wik22], which are also minimized by the regulator. The regulator has to find a balance between minimizing the state vector and the input vector. Higher values of the input vector result in faster reaching the set point by changing the state of the system faster, depending on the system it may be undesirable to have high fluctuations in the state vector. Later we want to regulate the amount of frames per second which are produced by the server and then displayed by the AR Device. As discussed later big fluctuations in frame rate result in bad user experience and therefore need to be kept minimal. On the other hand having a deviation of the state

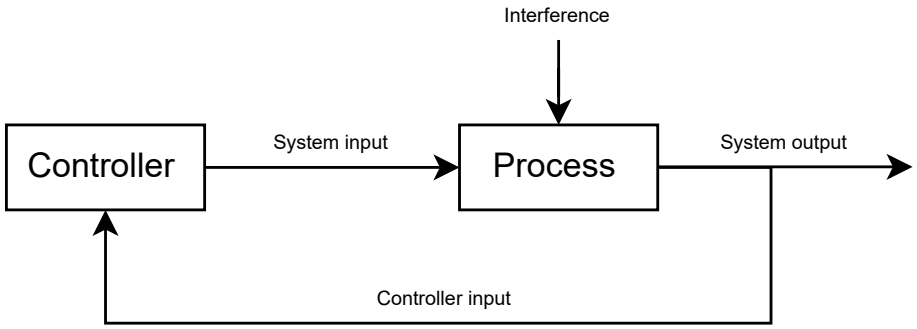

Figure 2.1: Feedback loop controller

to the set point is also undesirable. The balance between high input and high state vector values has to be found for each system individually. This shows that the LQ-Regulator is an optimization problem [Wiy].

The complete system for an LQ-Regulator is described by the formula:

$$\hat{x} = Ax + Bu$$

with A, B being constant matrices and x, u the state and input vectors respectively. A describes how the current state is affecting the next state. For example an inverted pendulum has a position and a velocity as current state. The next states position is affected by the current position and velocity. B does describe how the inputs to the system do affect the next state. The motor of the inverted pendulum is used for accelerating or decelerating the velocity in order to control the movement of the pendulum.

The goal of the LQ-Regulation is to calculate the gain matrix K, which then can be used to continuously calculate the appropriate new input to the system in order to converge the absolute value of state and input vector to zero. The formula for calculating the new system input is as follows:

$$\hat{u} = -Kx$$

$\hat{u}$ is the derivative of the new input. This means is has to be added to the last used input vector in order to apply it to the system.

The strength of the LQ-Regulator is the calculation of this gain matrix K because it can be controlled in an intuitive way. In order to calculate K the Riccati equation is used [YLM96], which indirectly gives the solution to K. As mentioned before the LQ-Regulator needs to find a balance between reaching the set point as fast as possible and avoiding huge input values (fast changed in the state). To tweak those parameters there are two more matrices Q and R used in the Riccati equation. The values in Q control how much high values in the state vector are penalized. The same is true for R but for the input vector.

Something that needs to be considered is that Q and R both need to be positive definite or else the equation will reward high values in x or u. This can be achieved by using a diagonal matrix with only values greater than zero on the diagonal.

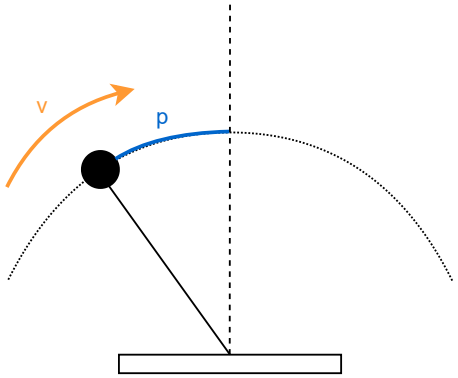

Figure 2.2: Inverted pendulum for LQ-Regulator

If choosing diagonal matrices each row of Q and R correspond to the same row in x and u respectively. Higher values in the Q or R matrices means a higher penalty for the corresponding value in x or u. The absolute values are not that important. Important are the relative values to each other. If choosing values other than on the diagonal, combinations of values can be punished. But as mentioned above it then has to be ensured that the matrices are positive definite.

LQ-Regulator Example

The following example is inspired by [Wür]. As an example we show here a simplified inverted pendulum that is controlled by an LQ-Regulator to maintain a steady zero position. The state of the inverted pendulum can be described by two attributes. The position relative to the set point in m (negative values corresponds to position on the left and positive values to the right) and the current velocity in m/s of the pendulum (positive values correspond to velocity to the right and negative values to the left). Figure 2.2 shows the setup for the inverted pendulum. The matrix A and state vector x for this example look as follows:

$$A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \qquad x = \begin{pmatrix} p \\ v \end{pmatrix}$$

The A matrix describes how the current state affects the next state. Therefore in order to explain A it is sufficient to look at the first part $\hat{x}_L = Ax$ of the LQ-Regulator formula $\hat{x} = Ax + Bu$. The used A matrix results in the following $\hat{x}_L = Ax$.

$$\hat{x}_L = Ax = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} p \\ v \end{pmatrix} = \begin{pmatrix} 0 \cdot p + 1 \cdot v \\ 0 \cdot p + 0 \cdot v \end{pmatrix} = \begin{pmatrix} v \\ 0 \end{pmatrix}$$

Since $\hat{x}$ is only the differential of the next state it has to be added to the current state in order to calculate the next state. With the shown A matrix the next state of the inverted pendulum based on the current state would be

$$x_{t+1} = x_t + \hat{x}_L = \begin{pmatrix} p \\ v \end{pmatrix} + \begin{pmatrix} v \\ 0 \end{pmatrix} = \begin{pmatrix} p + v \\ v \end{pmatrix}$$

This formula shows that the pendulum will move in one step the amount of the current velocity and the velocity will stay unchanged. This is a simplification because losses and gains due to e.g. friction and gravity are not considered. This could be added in case of the friction by changing the A matrix which would result in a friction reduced velocity in every step:

$$\hat{x}_L = Ax = \begin{pmatrix} 0 & 1 \\ 0 & -0.05 \end{pmatrix} \cdot \begin{pmatrix} p \\ v \end{pmatrix} = \begin{pmatrix} 0 \cdot p + 1 \cdot v \\ 0 \cdot p - 0.05 \cdot v \end{pmatrix} = \begin{pmatrix} v \\ -0.05v \end{pmatrix}$$

$$x_{t+1} = x_t + \hat{x}_L = \begin{pmatrix} p \\ v \end{pmatrix} + \begin{pmatrix} v \\ -0.05v \end{pmatrix} = \begin{pmatrix} p + v \\ 0.95v \end{pmatrix}$$

Going forward in this example we will use the simple version of A that ignores friction and gravity.

The B matrix and input vector u for the inverted pendulum is

$$B = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad u = \begin{pmatrix} U \end{pmatrix}$$

The input vector contains the voltage that is applied to the motor that is used to stabilize the inverted pendulum. A high voltage should increase the velocity if the pendulum is moving to the right and a negative value should decrease the velocity.

For the input matrix again only a part of the regulator formula can be used to explain the values. For the input matrix this is the last part $\hat{x}_R = Bu$.

$$\hat{x}_R = Bu = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} U \end{pmatrix} = \begin{pmatrix} 0 \cdot U \\ 1 \cdot U \end{pmatrix} = \begin{pmatrix} 0 \\ U \end{pmatrix}$$

Again this $\hat{x}_R$ has to be added to the current state to get the next. The next state will therefore be

$$x_{t+1} = x_t + \hat{x}_R = \begin{pmatrix} p \\ v \end{pmatrix} + \begin{pmatrix} 0 \\ U \end{pmatrix} = \begin{pmatrix} p \\ v + U \end{pmatrix}$$

The complete formula for the system that the Regulator will try to optimize is then

$$x_{t+1} = x_t + \hat{x}_L + \hat{x}_R = \begin{pmatrix} p \\ v \end{pmatrix} + \begin{pmatrix} v \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ U \end{pmatrix} = \begin{pmatrix} p + v \\ v + U \end{pmatrix}$$

which fulfills the two requirements: The pendulum will move in the direction of the velocity and also increase the velocity according to the voltage applied to the motor.

The following diagrams in Figure 2.3, Figure 2.4 and Figure 2.5 are showing how an inverted pendulum system with the before described matrices behaves. This images show the effect of the control input U to the velocity v and the position p. The goal is to regulate the position to zero.

The cost matrices Q_1 and R_1 are chosen as follows:

$$Q_1 = \begin{pmatrix} 100 & 0 \\ 0 & 0.01 \end{pmatrix} \qquad R_1 = \begin{pmatrix} 100 \end{pmatrix} \qquad \text{shown in Figure 2.3}$$

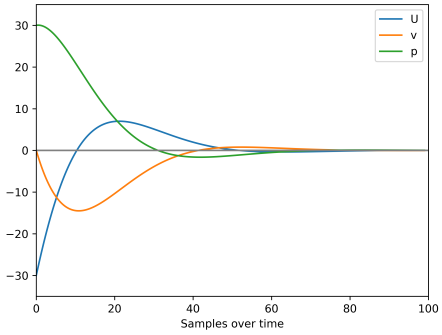

Figure 2.3: LQ-Regulator high p, U values, low v value

A high value in the Q matrix in the upper row and a low value in the lower row will punish the positional error more than high velocities. This means the regulator will try to reduce the positional error faster by sacrificing low velocity. The value in R is also high which means that the regulator will try to keep the voltage low.

With K calculated the next input value to the system can always be computed by the above mentioned $\hat{u} = -Kx$.

To show the effect of Q and R we also consider the following matrices

$$Q_2 = \begin{pmatrix} 10 & 0 \\ 0 & 0.01 \end{pmatrix} \qquad R_2 = \begin{pmatrix} 100 \end{pmatrix} \qquad \text{shown in Figure 2.4}$$

and

$$Q_3 = \begin{pmatrix} 100 & 0 \\ 0 & 0.01 \end{pmatrix} \qquad R_3 = \begin{pmatrix} 1 \end{pmatrix} \qquad \text{shown in Figure 2.5}$$

All runs of the LQ-Regulator were started with the initial state vector

$$x = \begin{pmatrix} 30 \\ 0 \end{pmatrix}$$

Comparing Figure 2.3 and Figure 2.5 the effect of a low versus high value of R can be seen. In Figure 2.3 where R has a high value the regulator tries to keep the absolute value of the blue curve low compared to Figure 2.5. This results in p reaching the set point earlier in Figure 2.5 because high values of U are punished less.

Comparing Figure 2.3 and Figure 2.4 show the effect of a lowering the control value in Q for p. This means that R is the highest value which means high values of U are being avoided at the expense of higher values of p and v. Because the U (voltage of the motor) is kept low the velocity changes slow and the position reaches the set point slower due to lower acceleration.

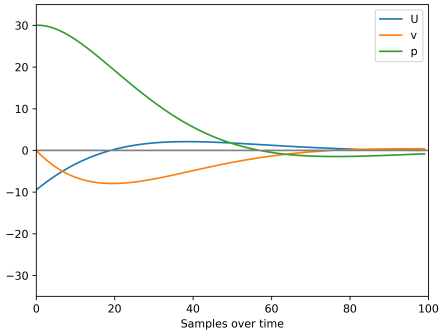

Figure 2.4: LQ-Regulator high U value, middle p value and low v value

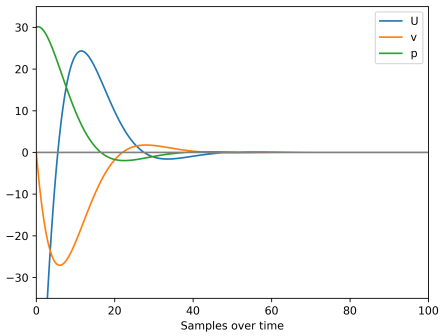

Figure 2.5: LQ-Regulator high p, low v and U value

2.1.2 Publish/Subscribe and MQTT

The approach later is a cyclic event-driven system which has multiple components that need to communicate with each other. The communication is implemented with a publish/subscribe system with MQTT as messaging middleware. This subsection will give an overview to better understand the system architecture and communication between the components.

Publish/Subscribe Pattern

There are two established ways to exchange messages in an distributed system. In the client server pattern there exist two components. The first component called server acts as provider of data or services and the other component called client can request data or services from the provider. The other possible way is event-driven message exchange with the publish/subscribe pattern. Here devices can publish events and other devices can get notifications to events that they are interested in. The client/server pattern has the following drawbacks, which is why we are using publish/subscribe later:

1. The server/client pattern or model is meant to be uni directional [Olu14]. The client makes a request to the server which provides the information to the client. It is not intended that the server can send information to the client without a previous request, meaning that the

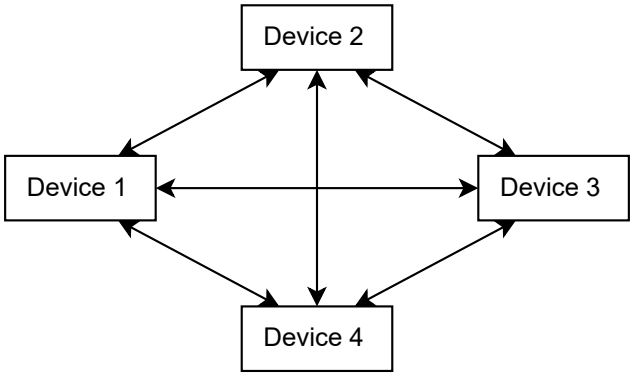

Figure 2.6: Client server pattern

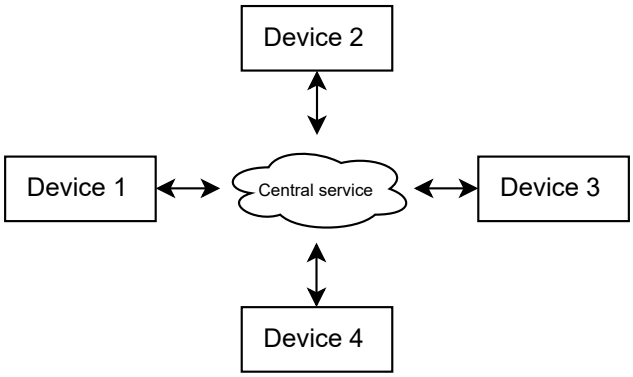

Figure 2.7: Pub/sub pattern

server is typically passive and waiting for requests and the client is actively making requests to servers. If a bidirectional communication is needed then both devices need to function as both server and client which complicates the system.

2. The amount of connections needed for the server/client pattern is not as scalable as the pub/sub pattern. The amount of connections needed in a client server pattern if every device should be able to communicate with every other device equals the amount of edges in a complete directed graph (directed graph because bi directional communication). The formula for the amount of connections is therefore $2 \cdot \frac{n(n-1)}{2}$ with n being the number of devices. This shows that the needed amount of connections scales quadratic with the amount of devices. In a pub sub system every device communicates with a logically central service (there may be distributed brokers or no brokers at all but from the point of view of the devices it is only one service) which forwards the messages as needed to the recipient devices. The communication channel between the central service and the devices may be bidirectional. This results in at most $2n$ required connections. Therefore the amount of connections scales linearly with the amount of devices.

The concept of pub/sub is that devices that are interested in some information can register that interest at a certain central service. Expressing the interest of getting a specific information is called subscription. Devices that express such an interest are called subscriber. Devices that want to provide information do so by publishing the information to the central service. Devices that publish

information are called publisher. A device can be subscriber and publisher at the same time. The central service that devices subscribe and publish to is called broker. It is responsible for receiving published content and forwarding it to all subscribers that subscribed to that content.

Publish/subscribe has four main advantages in an distributed system [EFGK03]:

1. *Space decoupling:* The devices that communicate with each other are not necessarily aware of one another. The device that has new information publishes that event to the broker but is not aware which subscribers want to receive that information. On the other hand the subscribers only get events from the broker and are not aware of the devices that published the information originally. This allows for an easier to manage system since all device only need to communicate with one device, the broker, instead of keeping track of all devices that need to be informed about a specific information.
2. *Time decoupling:* Devices do not need to be reachable at the time of the communication. If a subscriber is not available at the time of publishing the information will just be lost for that one subscriber or be forwarded later, depending on the implementation. In a client/server system the sender and receiver need to be connected.
3. *Synchronization decoupling:* The publisher will not be blocked while waiting for the receiving application to be ready to accept the connection. The publisher just publishes the information to the broker which then forwards the information possibly asynchronously (e.g. if the messages are buffered and send in batches) to the subscriber.
4. *Messages needed:* There is no theoretical limit of subscribers of a topic. This means with only one send message of the publisher a message could be received by an arbitrary amount of subscribers in contrast to the client/server system where a message for each recipient is needed.

There are three ways to implement the addressing of the topics in a pub/sub system:

1. *Topic based:* In this version the distribution is addressed by strings. The clients publish or subscribe to a topic string e.g `room1/temperature`. All information published under this string reaches all subscriber that subscribed to this string.
2. *Content based:* In this version the information is distributed based on the content of the message. There are different ways to evaluate the content. One way is providing a regex for subscribing. If the message content matches the regex then the message is forwarded to the subscriber.
3. *Type based:* In this version the subscriber can subscribe to types like `temperature` or `light-intensity` and get all messages of the types that they subscribed to.

We are focusing on topic based publish/subscribe since we only want the servers to be able to communicate with the controller and vice versa. We do not need the filtering possibilities that can be used in content based or typed based publish/subscribe. In topic based pub/sub systems information can be published under a certain topic, usually a human readable text e.g. a room temperature could be published under the topic ”living-room/temperature”. Devices that want to be informed about information concerning the room temperature would then subscribe to the same topic ”living-room/temperature”. Every time a publisher publishes new information under the topic ”living-room/temperature” the broker will send every subscriber that subscribed to this topic the new information [HTS08].

MQTT

MQTT (Message Queuing Telemetry Transport) [Oas] is a lightweight implementation of a topic based publish/subscribe system. MQTT is a broker-based publish/subscribe implementation this means it consists of two main components. The subscriber and publisher are both called MQTT-Clients, they are the first main component. The second component is the MQTT-Broker acting like the broker described for pub/sub systems. After subscribing to a topic the MQTT overhead is only a few bytes per message, which means small overhead when creating the message and sending it.

MQTT has some distinctive features:

- *Retained messages:* Normally when a publisher publishes a message to the broker, the broker will forward the message to all subscribers. Clients that are not subscribed to the topic at the time the broker forwards the message will miss out that information. MQTT has the feature to save the last received message for a topic at the broker. This feature can be used by setting a flag in the publish message sent by the publisher. The retained message at the broker is not necessarily the last sent message of a topic. The broker will retain the last message that had the retain flag set. If a client subscribes to a topic with a retained message than the broker will forward the current retained message to the subscriber. There can be at most one retained message per topic at a given time.
- *Quality of service:* By default messages are send only once regardless if the subscriber is receiving the message or not e.g. because of connection loss. MQTT provides a solution for more reliable messaging called ”Quality of Service” or QoS. This quality of service has three levels. Level zero ”at most once” just sends the message and does not check whether or not the message actually arrived. Level one ”at least once” ensures with a handshake that the message is received at least once by the receiver. There can be message duplicates. Level two ”exactly once” has two handshakes that ensure that the message arrives exactly once at the receiver.

2.2 Related Work

The closest related research field is the synchronization of real time media streams. These media streams can contain multimedia. Often there are video streams synchronized to audio streams on multiple devices. This is relevant for e.g. video calls. Multimedia has to be synchronized continuously due to the following dynamic factors [AH91]:

- Time of the transmission start as well as the transmission delay may vary
- Used compression technology works different for different media. E.g. a video stream might need to first process some packets in order to be able to display frames whereas audio streams may be playing as soon as received
- Processing speed of the sender and receiver might vary which will result in a desynchronization in the long term
- System resources might be dynamically taken by other programs and therefore slow down one device

2.2.1 Overview of Multimedia Synchronization Methods

There are several solutions to synchronizing media streams mentioned in [AH91]:

1. Vary output hardware I/O rate
2. Interpolate data
3. Skip or pause the data

Vary Output Hardware I/O Rate

This approach increases or decreases the data output rate depending on the state of the stream at hardware level. If a stream is ahead then the output rate is decreased if it lacks behind the output rate is increased. The disadvantage of this approach is that there is only one point of adjustment. This means multiple parallel streams on the receiver device (e.g. video and matching audio) cannot be adjusted independently. Our approach is based on this technique. We are increasing or decreasing the output rate of the server at the hardware level by adjusting the CPU frequency up or down and therefore increasing or decreasing the output rate of a specific server. Because we only have one output stream per server we are not effected by the disadvantage of this method.

Interpolate

This approach creates new intermediate data that interpolates between old and new data if there is more data needed than provided. The disadvantages of this approach is that there is data outputted that was not originally part of the stream and can lead to unwanted artifacts. It also needs more computational power.

Skip or Pause

This approach is to skip packets if a data source is lacking behind in order to catch up to the other one or to wait for the other data source. This comes with the disadvantage that a bigger buffer is needed because data needs to be stored until it is decided to skip it.

Buffering

Buffering can compensate for dynamic delays by playing out from the buffer at constant rate. The buffer can be filled at a non-constant rate. This means a buffer provides the advantage of decreases jitter at the expense of increasing the delay.

Synchronization Based on Network Clocks

The proposed synchronization mechanism in [EPD94] relies on synchronized network clocks that operate within a few milliseconds of each other. The packages sent are be timestamped by the servers. If the servers and the client have synchronized clocks then the client can calculate which server is how much time ahead. With this information the servers that have not the longest network delay can wait so they reach the maximum calculated delay. This way all packages arrive at the same time. We want to use an edge cloud that is possibly composed of consumer grade hardware which is not synchronized easily within a few milliseconds of each other which is why we cannot use this method.

Master/Slave Streams

[RVR92] proposes to categorize the streams that should be synchronized into two groups. One stream acts as ”master” stream and all other streams act as ”slave” streams. The master stream produces an uncontrolled amount of data. It is then measured how much each slave stream is lacking behind or is ahead. The streams are then independently sped up or slowed down (controlled data output) in order to synchronize to the master stream. This approach is useful if there is a reason to prioritize the continuity of one stream. In an audio podcast which is streamed with a complementary background video, the audio has priority in terms of streaming quality. In this case is does not matter if the background videos skips a frame or has to wait a few frames. Inconsistent audio however would be noticed. This is why the audio would be chosen as master stream and would just play back and the video would be the slave stream that is controlled to meet the speed of the audio. In our case we do not have a clear server that we can prioritize since all servers create an equally important partial information. However there are similarities between this approach and the one suggested in this work. We are creating a virtual master stream by averaging the current state of all servers. Then we synchronize all servers to this created virtual master stream. This way we are not prioritizing one server but give each server the same magnitude in the synchronization process.

3 System Model and Problem Statement

In this chapter important terms and assumptions are specified that will be used going forward in this work.

3.1 System Model

We want to display a complex simulation on a mobile device with only little computational power. Therefore the computation is outsourced to on-premise servers.

In our system we have three components:

1. A frontend device, in our case a resource-constrained mobile device (VR/AR devices, handheld device, etc.), that shows data to the user
2. Multiple edge cloud servers that help the frontend device to calculate data
3. A communication network between the servers and the frontend device

The mobile device can display images in real-time to the user. Possible devices could be Head-Mounted-Devices (HMD) which are headsets that obstruct the full field-of-view of the user except a screen (which may me transparent) that is build directly in front of the eyes of the user. The main goal with this devices is to display interactive content in real time. The image displayed to each eye can differ which makes displaying 3D stereoscopic content possible.

There are two general types of HMD:

1. The HMD is connected via wire that is plugged into a computer which is used to calculate all the needed images that are displayed on the screens (used as peripheral device). This type of HMD has three tasks. The first task performed by the HMD is collecting the needed sensor data for positioning the virtual screen. The second task is sending this data to the computer which will calculate the frame. And the last task is to display the calculated frame.
2. The HMD is not connected via wire that is plugged into a computer. This means all sensor data is collected and processed on the mobile device. After processing the sensor data the displayed image will also be computed on the HMD. These devices are powered via an internal usually rechargeable battery. Data can be sent via a WiFi connection from and to an on site edge cloud, but the device could also work without.

We are focusing on the latter type since scaling the computational power of a connected computer can be done easily by increasing size and amount of computing units (e.g. graphics cards) inside. The possibility of increasing the computational power of the standalone AR-/VR-device is limited since the computational power is mainly restricted by the power output of the battery. This can not be increased in an arbitrary amount because the user has to carry it in order to stay mobile. Because

"batteries are the largest single source of weight in a portable computer"[FZ94] the headset would become at some point too heavy to carry. Putting bigger graphics chips in the headset also becomes infeasible at a certain weight.

The servers which are used to calculate the simulation are in the same network as the mobile device. They are at most one hop away. To simplify the problem it is assumed that there is no network delay between the server(s) and the mobile device. The computational power of the servers can be increased or decreased as needed. Each server will be used for horizontal scaling of the complex remote computation, such as a simulations. The servers will compute information and then send it to the mobile device which will display it.

Our system is shown graphically in Figure 3.1.

We assume a set of servers from one to n seen on the left in Figure 3.1. A single server is referenced as S_i.

One frame has multiple calculation jobs that are calculated distributed on the edge cloud servers. After a calculation on S_i is finished it will send a data packet ${DP^{(S_i)}}_T$ that contains all the partial information needed for the mobile device to render the complete image once all data packets for a frame arrived (orange arrows in Figure 3.1). After that, the mobile device renders the frame and displays the resulting content to the user.

We explicitly do not assume that every server is built with the same hardware and has the same programs running in the background. This heterogeneity in hardware and software results in a non constant calculation speed of one frame. The complexity of calculating a frame may also vary, contributing to a variable time that is needed to calculate a frame on a specific server.

The calculation speed of a server may vary for a variety of reasons, for example:

1. Other programs want to use the CPU
2. Complexity of the calculated frames is not constant
3. CPU can not sustain a specific speed due to thermals

The servers have some limitations in terms of CPU frequency scaling. We assume that the frequency of each CPU can be scaled dynamically. The granularity of scaling can vary. For instance, x86_64 servers often allow for scaling by setting the CPU frequency within a given range. The CPU frequency can only be set to discrete integer MHz values. They also cannot be set to arbitrary values. We assume an interval to choose valid CPU frequencies from.

The mobile device will send periodically information about the current synchronization state in an information packet IP_T to a controller (black arrow in Figure 3.1). The controller will process the packets and decide what server needs to be speed up and which needs to be slowed down. This information will be sent to each server with a control packet ${CP^{(S_i)}}_T$ (blue arrows in Figure 3.1). Which will react accordingly to the information contained in ${CP^{(S_i)}}_T$.

To summarize these are the assumption that are made about our proposed system:

1. The CPU clock speeds of all servers are adjustable at runtime
2. The servers, mobile device and controller are in the same local network, therefore the networking delay is not considered in this work

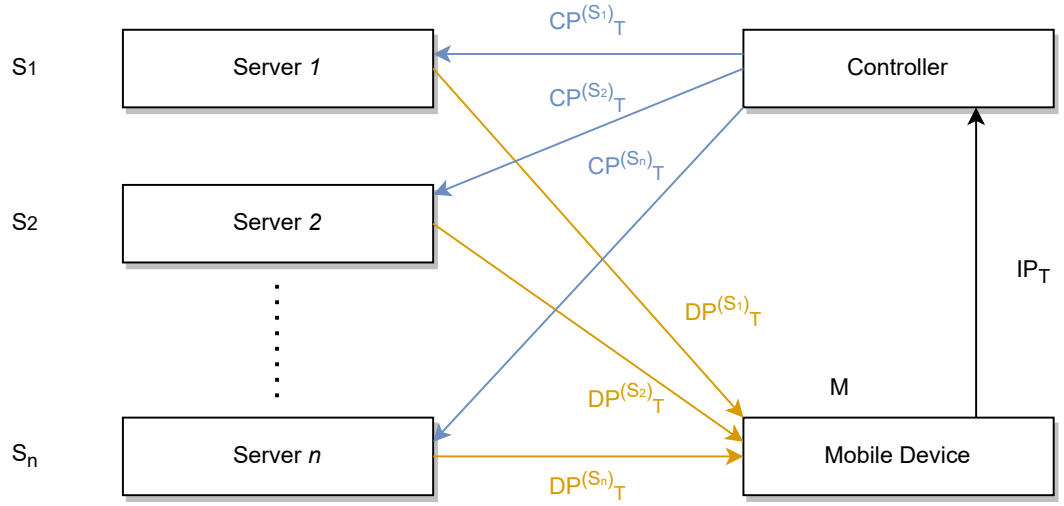

Figure 3.1: System model

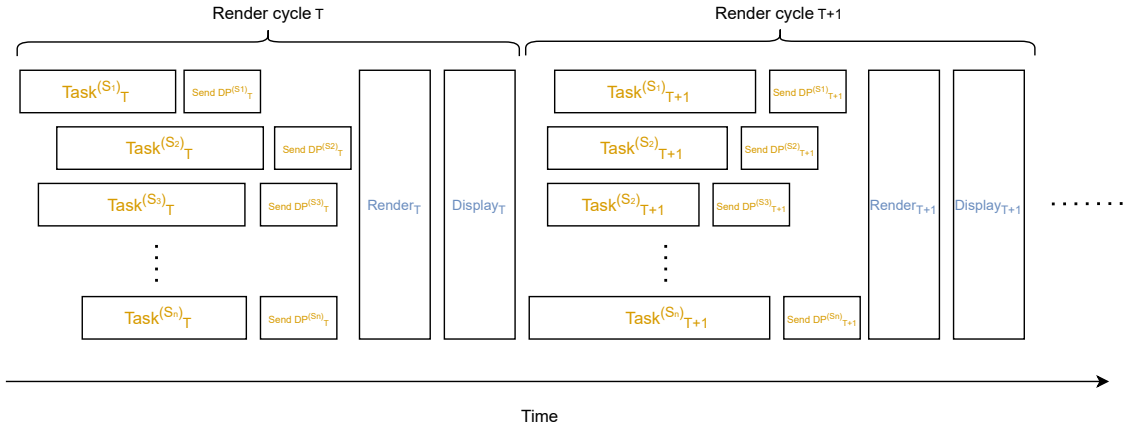

Figure 3.2: Rendering cycles

3. The mobile device only renders the information provided by the edge cloud servers which may distribute the calculation load between each other. The computational power of each server is limited and heterogeneous which is why multiple servers may be used to increase the total computational power

The rendering timeline for the mobile device is shown in Figure 3.2. The mobile device can only render a frame and display it after all packets of a frame arrived. It continuously waits for new packets and renders and displays them after all packets of a frame arrived.

3.2 Problem Statement

This section describes the problem that will be solved in this work.

A problem with HMDs is that they can induce motion sickness to the user and therefore the "associated symptoms of headache, postural discomfort and unsteadiness" [GG05].

There are two factors that are contributing to a user getting motion sick. The first factor is the delay of the images that are displayed (we need low latency). With higher latency the images that the eyes receive do not match the physical movement of the head, disturbing the vestibulo-ocular reflex. The body tries to compensate the movement of the head by moving the eyes in opposite direction to stabilize the retina [CKY20], because the actual field of view does not change due to latency the brain gets confused. Though is has to be noted that there is conflicting evidence about the affect of latency on motion sickness [DL97] [DVFG01]. Draper et al. suggests that a user can adopt to the longer delay between head movement and visual changes if the delay is constant (up to 250 ms).

This shows the second important factor to reduce motions sickness. In order for the brain to be able to adjust for the increased latency, it must stay constant. The brain is not able to adjust for fast changes in latency. This is why images must also be displayed at a consistent rate [CKY20].

Long delays however lead to bad user experience at best, and one might become motion sick at worst, which is why we want to minimize delay.

A high refresh rate, meaning images per second, will improve the user experience and is therefore also important.

A possible solution would be to increase the computational power of all servers to their maximum (e.g. running always with the highest possible CPU clock speed). We assumed that the servers, running the simulation have heterogeneous hardware. The faster server on full speed will compute frames faster than the others. The additional frames however cannot be used because only a complete image is displayed and therefore the server produces more frames than needed, consuming more energy than necessary.

We want to find a synchronization state which fulfills the following criteria:

1. Frame rate of all servers should be equal.
2. Frame rate should be maximized under requirement one. That is, the slowest server and/or most complex computational task will define the frame rate that others have to adjust to.
3. The delay should be minimized. Therefore, we would like to avoid any approach based on buffering more than one frame and delaying frame display. Instead, frames should be displayed as soon as possible after they have been received. This is achieved by the servers outputting all ${DP^{(S_i)}}_T$ at the same time.
4. Frame rate should be constant (low variance in frame rate).

Part of this work is therefore finding parameters to increase or decrease the processing speed of the frame calculation to control this synchronization state.

Because of the assumption that the calculation is not running exclusively on the system the calculation time is dynamic, this means it is not sufficient to synchronize the servers once, but a control mechanism is needed in order to keep the servers in a constant synchronized state. This control mechanism needs a sufficient architecture in order to be effective. The goal of this work is to develop a control mechanism, implement it and evaluate its effectiveness.

In order to prove feasibility of the developed approach a proof-of-concept implementation for a synchronization mechanism for a distributed system of edge cloud servers will be shown in this paper. This proof-of-concept implementation will then be evaluated. The evaluation is done by measuring the degree of synchrony that is be achieved with the proposed system.

4 Approach

To solve the problem of synchronizing multiple servers that are calculating parts of a frame we propose a strategy that uses an LQ-Regulator to speed up or slow down calculations in order to meet the synchronization criteria defined in Section 3.2. To verify the feasibility of our approach we engaged in a preliminary study that is supposed to test on a modeled environment if the LQ-Regulator is usable for regulating servers by modifying the CPU frequency. After this we show how our approach is working.

4.1 Preliminary Study

As described in Section 3.1 the complexity of calculating a frame on a specific server may vary from frame to frame. This is why it is not sufficient to synchronize the servers once but a system must be developed that tries to keep the servers constantly in a synchronized state. The synchronization must therefore be performed in regular intervals. As already introduced Chapter 2 there are already established concepts called regulators.

Such a system consists of three components Figure 2.1:

- The process that needs to be regulated
- The controller itself
- Some interface that allows to change the behavior of the regulated component via actuators

The sensors observe the process and send samples to the controller. This controller then decides based on predefined rules in which way the behavior should change in order to get the desired behavior of the regulated component.

In our case the component that should be observed is the combination of all servers, we want them to output:

- Frames with high frame rate
- A constant frame rate (low variance)
- All ${DP^{(S_i)}}_T$ at the same time

We can observe certain characteristics based on which we can determine the state of synchronicity. The synchronization state is quantified as follows:

1. The number of frames per second can be measured by counting the amount of frames that a server outputs per second. Alternatively it can be calculated by measuring how long it takes the server to calculate one frame. Assuming the server takes approximately the same amount of time per frame the number of frames can be extrapolated based on this information. This can be done by dividing second by the time it took to calculate one frame: $\texttt{fps}(x) = \frac{1000}{x}$ with x being the calculation time for one frame in milliseconds. It is possible to use the calculation time instead of frames per second. We are choosing frames per second. We aim to maximize the amount of frames per second of all servers.

2. The variance of the frame rate can be quantified as follows:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

   We calculate the variance over a time window of frames. We form the variance only over the latest n calculations because we want to evaluate the current synchronization state, for this it is not relevant e.g. how the variance was ten minutes ago. The variance is then calculated with:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(\texttt{calculation\_time}[\text{i}] - avg(\texttt{calculation\_time}))^2$$

   A higher variance means more variability in the calculation time, this means we want to keep the variance as low as possible. A variance of 0 is the best achievable value, meaning the calculation time is constant. We aim therefore for a variance of 0.

3. To check if the server is behind or ahead with producing frames we need to store the index (sequence number) of the last received data packet per server. By calculating the subtraction last received packet index of a server minus the average of all last received indices over all servers it can be determined if S_i is behind or ahead. If the value is negative it is behind and if the value is positive it is ahead. If this metric for all servers is zero it means that all servers are calculating the same frame in this moment. We aim therefore for a value of 0 for all servers. The formula for this calculation looks as follows:

$$\texttt{packet\_error}[\text{i}] = \texttt{latest\_packet}[\text{i}] - avg(\texttt{latest\_packet})$$

The IP_T contains two of the three pieces of information as comma separated values for the controller which splits the values into a vector (explained in detail later) for the regulator to decide if a server needs to produce more or less frames per second. The regulator needs to have information about the frames per second and the current packet index. The variance is not needed as input for the controller since it self controls the variance. Changing the calculation speed often and more significant, results in a higher variance while changing it less often and less significant, results in lower variance. We will go later into detail on how we control the variance with the LQ-Regulator.

In order to evaluate the feasibility of the regulation strategy a model environment was created to test the regulator.

The calculation of a neural network which would be calculated on a server is substituted by a generic way to produce load on the CPU. We calculated Pi to a certain decimal point.

Increasing the number of decimal points calculated will not result in a linear increase of the time needed to calculate it as shown in Figure 4.1. Later we want to simulate the processing delay (combination of task complexity and CPU power, including frequency) with the amount of pi decimals calculated. This is why we want the increase of the decimal points to behave similar to a consumer grade pc/server in terms of CPU frequency. To do this we calculate pi not to the x decimal point but the $\sqrt{x}$ decimal point. This near linear calculation time can be seen in Figure 4.2. this is meant going forward when talking about the number of decimal points calculated unless otherwise stated.

As base we calculated Pi to the $\sqrt{10000}$ decimal point. Then to simulate changing complexity and time it needs to calculate a frame we added or removed decimal points that were calculated at run time. This changes the time needed for calculating pi.

If we increase the amounts of pi decimals calculated the complexity increases, if we decrease them the complexity decreases. This changing amount of pi decimals that is calculated we call `calculation_complexity`. Since a decreased complexity means higher frame rate the `calculation_complexity` behaves inverse proportional to the amount of calculated frames per second as shown in Figure 4.3 and Figure 4.4.

This `calculation_complexity` simulates the heterogeneous complexity of different tasks and speed of different machines of the neural network and we have to treat it as given since we cannot control the complexity of the real system.

Because of this we need a `control_offset` which we will use to simulate the CPU frequency. The `control_offset` is added to the `calculation_complexity`. The value that is calculated is then $\sqrt{\texttt{calculation\_complexity} + \texttt{control\_offset}}$ pi decimals.

We have to keep in mind since we want to simulate the CPU frequency with the `control_offset`, that increasing the CPU frequency will result in more computational power and therefore more frames per second. An increase of the `control_offset` however will decrease the frame rate. This will be compensated later in the actual approach by negating the parameter that controls the `control_offset` or CPU frequency respectively.

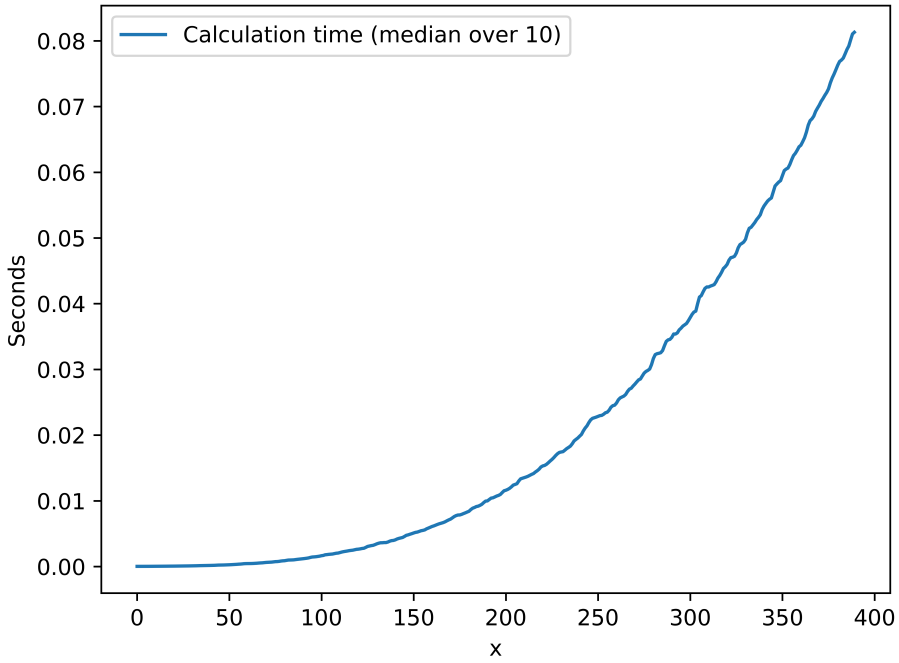

Figure 4.1: Calculation time of pi to the x decimal point

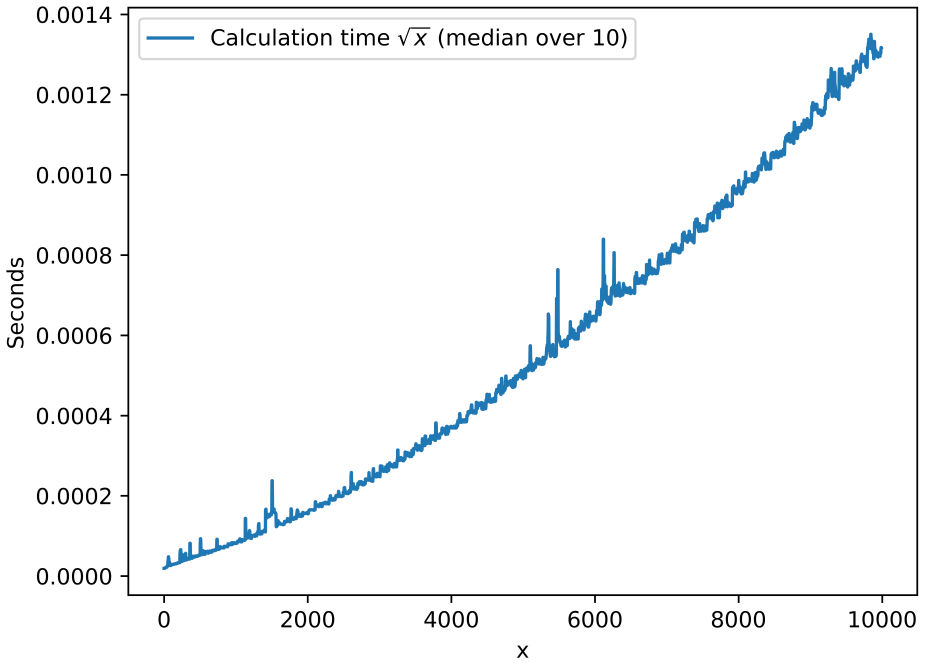

Figure 4.2: Calculation time of pi to the $\sqrt{x}$ decimal point

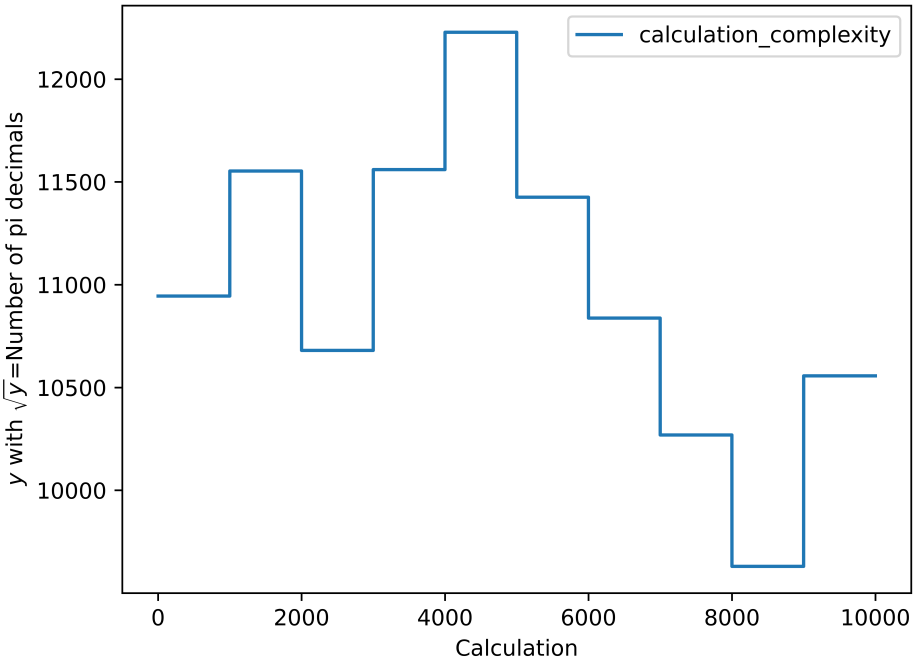

Figure 4.3: Calculated pi decimals

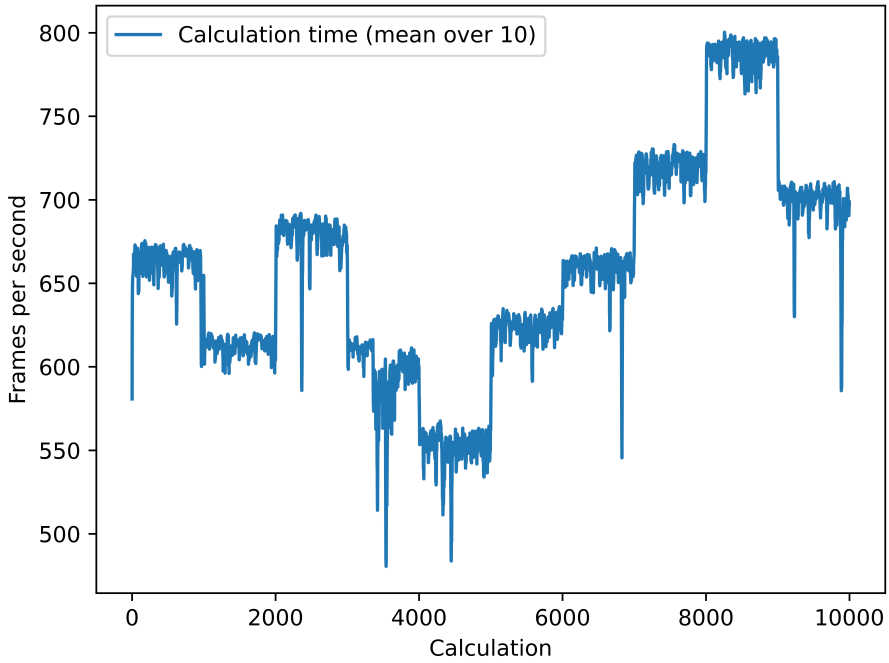

Figure 4.4: Resulting frame rate of calculated pi decimals

4.1.1 Synchronizing one Server to a Fixed Frame Rate

The first test was to synchronize one model server to a fixed frame rate. This includes two of our synchronization requirements. The frame rate and the variance of the frame rate. Because the LQ-Regulator needs a state vector that it can optimize to we need a set point for the frame rate, later we need to find the maximum frame rate that all servers can regulate to. Then we calculate the error (current frame rate minus set point) and use this as input for the regulator. The variance is not part of the input. As input vector of the LQ-Regulator we use the change of the `control_offset`.To control the variance we increase the value of the R matrix which has only one value in this case. As discussed in Section 2.1.1 a high value in R means that high values in u are punished more. Because u is the change in `control_offset` keeping this value low means that the calculation speed and in extension of this the frame rate changes slowly. This results in a lower variance.

As base we are calculating pi to the $\sqrt{17000}$ decimal point. The simulation of the changing complexity can be seen in Figure 4.5 (system disturbance). As `calculation_complexity` we used $\sqrt{17000 + y}$ with y the values of Figure 4.5. The complexity changes step-wise three times while calculating 10000 simulated frames. After 2500 frames calculated the `calculation_complexity` was increased by 5300. At 5000 frames it was decreased by 6000 and at frame 7500 was increased by 8000.

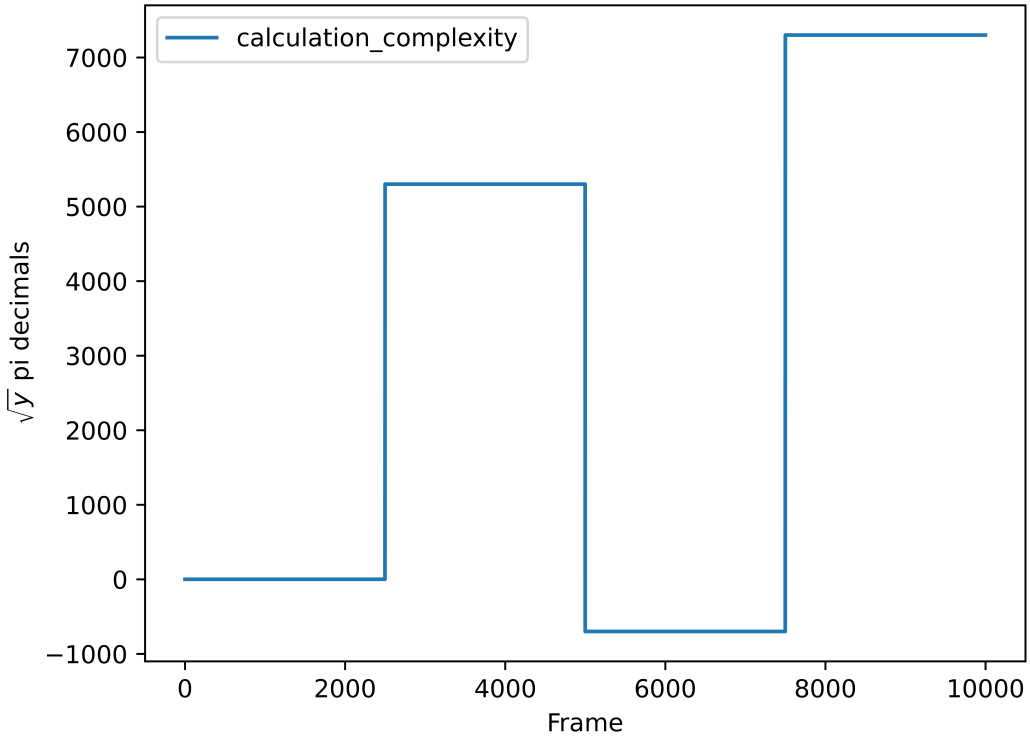

Figure 4.5: Simulated complexity change

The goal of the regulator was to regulate the `control_offset` in such a way that the model would produce frames at a rate of 60 frames per second. The LQ-Regulator matrices are A, B, Q and R as stated in Section 2.1.1.

For the state vector we chose:

$$x = \begin{pmatrix} \texttt{fr\_error} \end{pmatrix}$$

where `fr_error` was calculated as the deviation between the target frame rate (60 frames per second) and the mean frame rate over the last 10 frames. The mean frame rate is used here so the regulator would not try to compensate for single complexity spikes (low-pass filter).

As input vector we used the derivative of the `control_offset`:

$$u = \begin{pmatrix} \texttt{control\_offset'} \end{pmatrix}$$

This is the output of the regulator that we will feed back into the system.

The LQ-Regulator formula looks as follows:

$$\hat{x} = A \begin{pmatrix} \texttt{fr\_error} \end{pmatrix} + B \begin{pmatrix} \texttt{control\_offset'} \end{pmatrix}$$

Also we chose as system matrix A and input matrix B the following matrices which means that for the A matrix that the `fr_error` of the next state is the same as the current state (based on the current state) and the for the B matrix that the `fr_error` is increased or decreased one to one according to the change of the `control_offset`:

$$A = \begin{pmatrix} 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 \end{pmatrix}$$

As discussed in Section 2.1.1 the matrices Q and R will define how the regulator will react on the values of x and u. The values used in the tests are:

$$Q = \begin{pmatrix} 10 \end{pmatrix} \qquad R = \begin{pmatrix} 1000 \end{pmatrix}$$

and

$$Q = \begin{pmatrix} 1000 \end{pmatrix} \qquad R = \begin{pmatrix} 10 \end{pmatrix}$$

As shown in Figure 4.6 the LQ-Regulator regulates the `control_offset` in such a way that it will stay stable around a value until the `calculation_complexity` changes and adoption is needed again. In this test we are only synchronizing to a fixed frame rate, this frame rate is achieved by a certain amount of pi decimals (assuming that the system has constant computational power) the regulator will in this case always try to optimize in such a way that this specific amount of pi decimals is calculated.

This can be observed in Figure 4.6 and Figure 4.5. After the `control_offset` converged against a value and the `calculation_complexity` increased at frame 2500 by 5300 the LQ-Regulator regulated the `control_offset` down by roughly 5300.

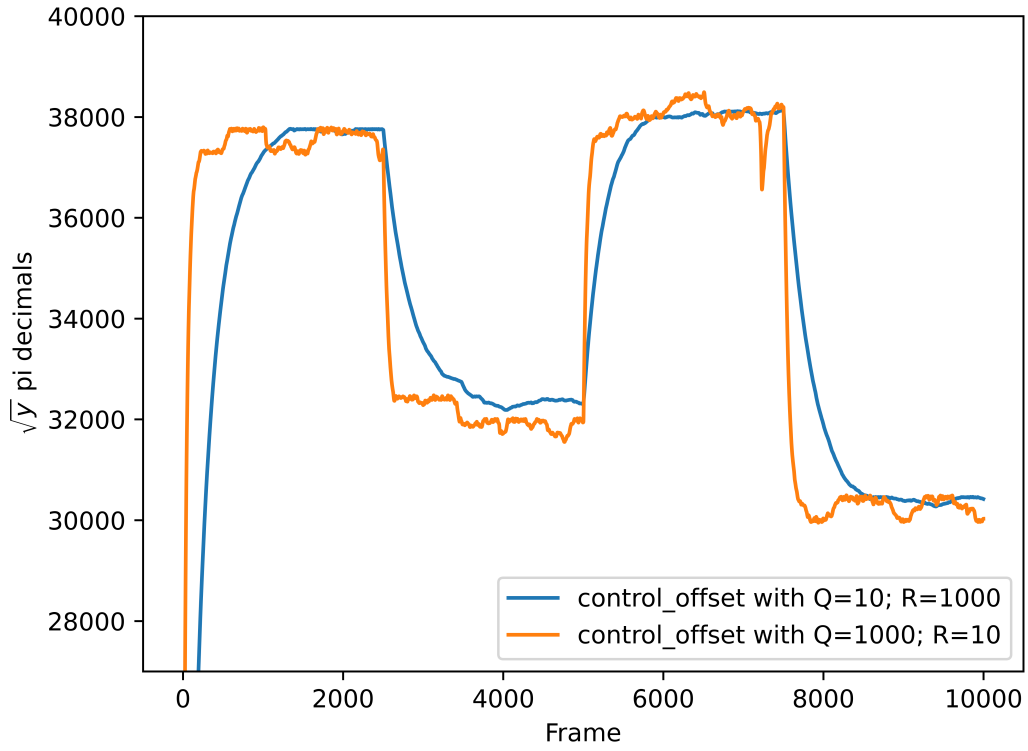

Figure 4.6: Regulation of the LQ-Regulator

Since this is the behavior that we would expect the regulator is working correctly with the chosen values.

Figure 4.6 shows the impact of changing the control matrices of the LQ-Regulator. It shows what impact a change of R and Q has on the regulated system. An increase of the values in R should punish big input values. For our system this means a big R values should result in a slower change of the `control_offset`. The blue line shows this by converging slower and also staying more constant once it reached the target value, because a change of the `control_offset` is discouraged by a high value of R. The system will converge regardless of the values in R and Q, they just control how fast the system will converge to the set point.

Figure 4.7 shows the resulting frames per second. The frame rate stays around the target frame rate of 60 frames per second. The only parts where it deviates from the target value is at the beginning where is needs to reach the set point for the first time and at the three times where the `calculation_complexity` changes and the regulator needs to adjust the `control_offset` in order to reach the 60 frames per second again. In this figure the different behavior of the regulator can also be observed, if the value of R is high than the blue line needs longer to reach the target frame rate.

Figure 4.8 shows the variance of the frame rate which stays under four apart from a few outliers (which are around 50) and at the points where the `calculation_complexity` changes and the frame rate needed to be adjusted.

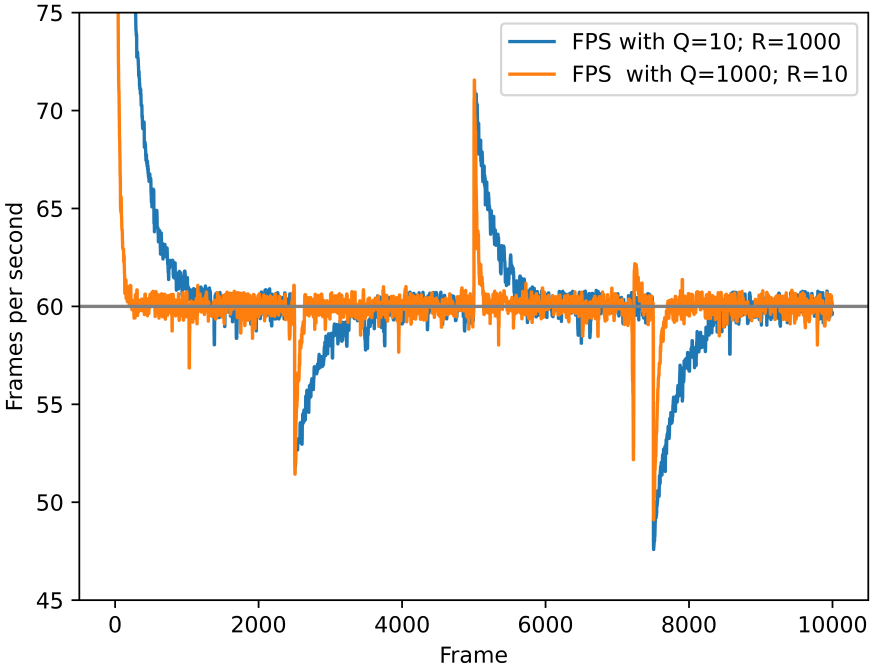

Figure 4.7: Resulting frame rate based on regulation of the LQ-Regulator

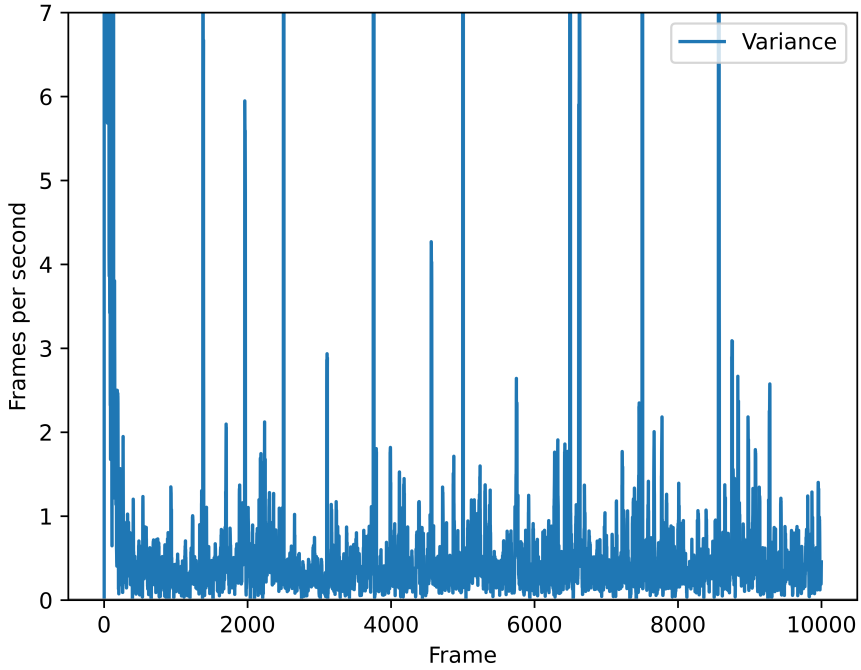

Figure 4.8: Resulting variance based on regulation of the LQ-Regulator

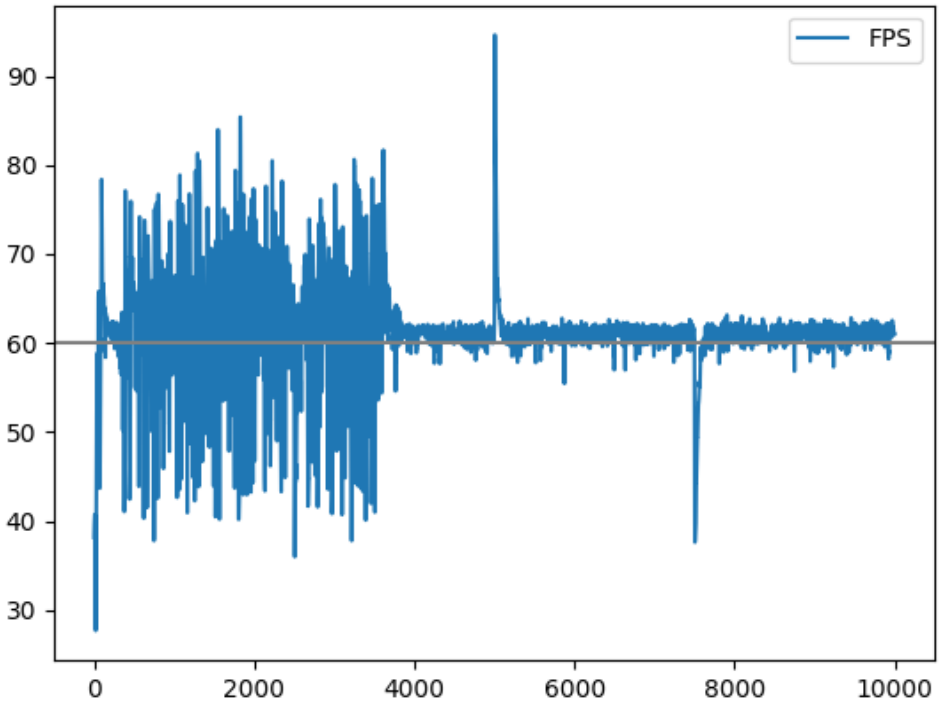

Figure 4.9: `control_offset` with activated power saving (automatic frequency scaling)

While running the tests we observed that the regulator can only regulate the system if it has enough control over the system that it wants to regulate. The tests where run on Windows 11. When turning on power saving of the operating system the regulator was not able to adjust the `control_offset` in a useful way. The power saving of the operating system acts comparable to a second regulator that regulates against our regulator. In Figure 4.9 you can see the point at which the power saving of the operating system was turned off (around frame 4000). The other two peaks in frame rate are the expected peaks from the changed `complexity`.

4.2 LQ-CPU-Frequency-Synchronization

Our preliminary studies showed that it is possible to synchronize a model server to a specific frame rate while trying to reduce the variance of the frame rate. With this we have two of our three synchronization requirements (high frame rate, low variance in frame rate and low variance in the last frame produced) fulfilled (first two are met, last one is still open).

To complete our approach the LQ-Regulator also has to consider the index of the last produced frame of each server. This has to be done in order to reduce the need of buffering on the mobile device. If all servers are producing the same frame at the same time then the mobile device can display the frames as they are calculated. If one server produces newer frames than another, then the mobile device will have to buffer the information until all partial frames of the servers arrived for a specific frame. To do this we average the index of the last produced frame of all servers. Then we speed up the servers which index of last frame is lower than the average and try to slow down the servers that have a higher index of last frame. This way when every server has an error of zero to the average index of last frame all of them are producing the same frame next.

The extra parameter that we want to synchronize results in new rows in the A and Q matrices of the LQ regulator.

4.2.1 Regulator for LQ-CPU-Frequency-Synchronization

The process of synchronizing multiple servers to each other by using an LQ-Regulator to speed up or slow down individual servers we call *LQ-CPU-Frequency-Synchronization*. The input vector u for the LQ-Regulator contains the change in CPU clock frequency for each server that should be controlled, called `control_offset`. The state vector contains the values that should be minimized in our case that is the error of frame rate to the set point and the error of the packet index of each server to the average of the last produced frame.

Formulas for the `fr_error`, `packet_error` and the `control_offset`:

1. `new_cpu_frequency_Si` = `old_cpu_frequency_Si` + `control_offset_Si`
2. `fr_error_Si` = `fr_Si` $- avg($`fr_Sx`$)$
3. `packet_error_Si` = `last_frame_produced_Si` $- avg($`last_frame_produced_Sx`$)$

The CPU freqency is measured in MHz. For n servers the resulting input and state vectors look as follows:

$$x = \begin{pmatrix} \texttt{fr\_error\_S1} \\ \vdots \\ \texttt{fr\_error\_Sn} \\ \texttt{packet\_error\_S1} \\ \vdots \\ \texttt{packet\_error\_Sn} \end{pmatrix} \qquad u = \begin{pmatrix} \texttt{control\_offset\_S1} \\ \vdots \\ \texttt{control\_offset\_Sn} \end{pmatrix}$$

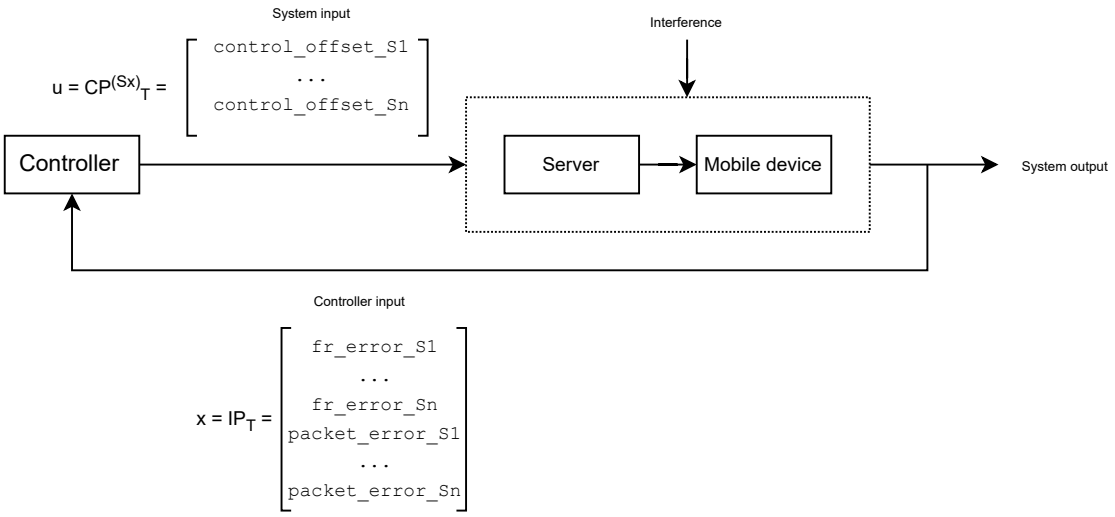

Figure 4.10: Feedback control loop adapted to our approach

Figure 4.10 shows the standard feedback control loop adapted to our system with the messages that are exchanged. The controller gets the current state from the mobile device via the information packet and calculates the new control input for the servers which is then send via the control packets to the servers.

With x and u defined the matrices A and B can be modeled. A defines how the current state affects the next state. The next error of the frame rate is not affected by the current frame rate error and packet error, meaning the entries of the state vector stay the same. The packet error values of the next state are also not affected by the current state. The matrix that is suitable for this case is the identity matrix.

B defines how the input changes the next state. If the CPU frequency is increased meaning the `control_offset'` is greater than zero then the amount of frames per second should increase. If the CPU frequency is decreased (negative `control_offset`) then the amount of frames per second should be decreased.

The same is true for the packet error. If the CPU frequency is increased than the server will produce more frames compared to before and therefore catch up with the other servers and vice versa. This means that the `control_offset` has the same effect on the packet error as on the frame rate error. A positive `control_offset` will increase the error and a negative `control_offset` will decrease it. To achieve this we choose a matrix that has the values of two identity matrices on top of each other in one matrix. This means that the control offset for a specific server increases the frame rate and packet error (can be negative) one to one to the control offset for this server. Using only ones in the matrix means that an increase of one MHz translates to exactly one frame per second faster which is a simplification. The correct constants would need to be empirically determined per server that reflect how the increase of CPU clock frequency affects the frame rate and package error. However we later show that the approach works correctly with this simplification. Further optimization may be possible by evaluating the correct constants through tests on the individual servers that will be

controlled by a specify LQ-Regulator. This means that a separate LQ-Regulator for each system (set of servers, controller and mobile device where only the servers are relevant for the regulator creation) needs to be created for optimal usage of this approach.

For n servers the matrices of the LQ-Regulator look as follows:

$$A_n = I_n \qquad B = \begin{pmatrix} I_n \\ I_n \end{pmatrix}$$

This results in the follow equation:

$$x_{t+1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \texttt{fr\_error\_S1} \\ \vdots \\ \texttt{fr\_error\_Sn} \\ \texttt{packet\_error\_S1} \\ \vdots \\ \texttt{packet\_error\_Sn} \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \texttt{control\_offset\_S1} \\ \vdots \\ \texttt{control\_offset\_Sn} \end{pmatrix}$$

$$x_{t+1} = \begin{pmatrix} \texttt{fr\_error\_S1} + \texttt{control\_offset\_S1} \\ \vdots \\ \texttt{fr\_error\_Sn} + \texttt{control\_offset\_Sn} \\ \texttt{packet\_error\_S1} + \texttt{control\_offset\_S1} \\ \vdots \\ \texttt{packet\_error\_Sn} + \texttt{control\_offset\_Sn} \end{pmatrix}$$

With these matrices it is defined *what* system should be controlled by the LQ-Regulator. To define *how* the LQ-Regulator will optimize the system the Q and R matrices are required. Since the frame rate error and the packet error are both in the state vector they are controlled by the Q matrix. With our two parameters we have three choices how we want the LQ-Regulator to act.

Q_1 Prefer low error in frame rate

Q_2 Anything between Q_1 and Q_3 ($w \in [10 - 1000]$) weighs between Q_1 and Q_3

Q_3 Prefer low packet error

$$Q_1 = \begin{pmatrix} 1000 \cdot I_n \\ 10 \cdot I_n \end{pmatrix} \qquad Q_2 = \begin{pmatrix} w \cdot I_n \\ (1010 - w) \cdot I_n \end{pmatrix} \qquad Q_3 = \begin{pmatrix} 10 \cdot I_n \\ 1000 \cdot I_n \end{pmatrix}$$

Note that the absolute values do not matter, relevant are the values relative to each other. With Q it is possible to accommodate for two of the three synchronization criteria: Low packet error and low frame rate error. In order to accommodate for the low variance in the frame rate the R matrix is used.

R controls how much the CPU frequency is changing and in extension the frame rate. A faster changing CPU frequency will result in a higher variance of the frame rate. To keep the variance low, high values in R needs to be set, this will result in lower values of u which corresponds to the

change in the CPU frequency (high values in R punish high values in u, meaning high values in R will therefore result in lower values in u). This will however cost slower reduction of the frame rate and packet error. We found that a value of 1000 for all values in R on the diagonal is a good compromise between responsive adaption to the set point and low variance. This means R looks as follows:

$$R = \begin{pmatrix} 1000 \cdot I_n \end{pmatrix}$$

The earlier explained Q_2 matrix was to show what effect the values in Q have. It is not required to set the values as showed in the definition of Q_2. We found that choosing Q as follows worked best for our setup:

$$Q = \begin{pmatrix} 600 \cdot I_n \\ 1000 \cdot I_n \end{pmatrix}$$

For three servers the matrices of the LQ-Regulator look as follow when choosing Q_3:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1000 \end{pmatrix} \qquad R = \begin{pmatrix} 1000 & 0 & 0 \\ 0 & 1000 & 0 \\ 0 & 0 & 1000 \end{pmatrix}$$

The state and input vectors used are:

$$x = \begin{pmatrix} \texttt{fr\_error\_S1} \\ \texttt{fr\_error\_S2} \\ \texttt{fr\_error\_S3} \\ \texttt{packet\_error\_S1} \\ \texttt{packet\_error\_S2} \\ \texttt{packet\_error\_S3} \end{pmatrix} \qquad u = \begin{pmatrix} \texttt{control\_offset'\_S1} \\ \texttt{control\_offset'\_S2} \\ \texttt{control\_offset'\_S3} \end{pmatrix}$$

To evaluate the approach we first applied it to the model environment of Section 4.1 and tried to synchronize three model servers using the pi decimals for simulating the CPU frequency. To do this we again had to use $-B$ for the LQ-Regulator because adding more pi decimals that need to be calculated behaves inverse proportional to an increased clock speed in terms of calculation time.

Figure 4.11 show the packet error in a system of three servers. The orange lines show how the system behaves if no synchronization mechanism is applied to it. The blue lines show how the system behaves when LQ-CPU-Frequency-Synchronization is applied to the system. In the LQ-CPU-Frequency-Synchronized system the lines stay close to set point whereas in the non-synchronized

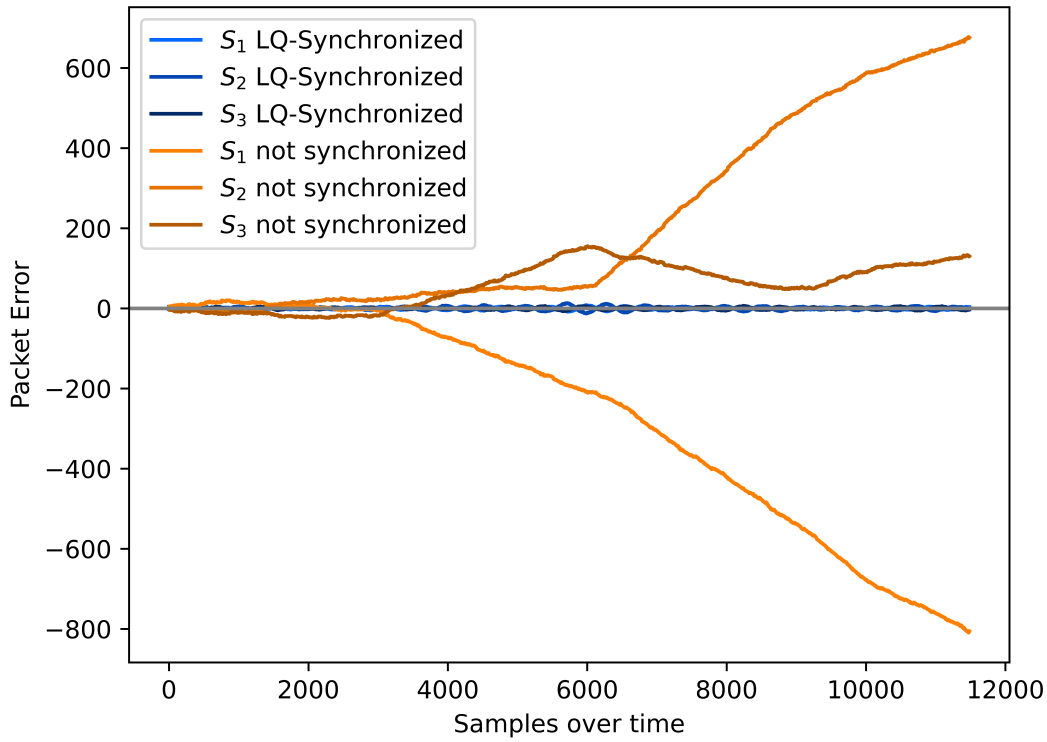

Figure 4.11: Packet error when synchronizing three model server

system the lines deviate more and more from the set point. The non-synchronized system has 90% percentiles for the packet error of 706.33, 600.66 and 127.33 for the individual servers. The synchronized system achieves 90% percentiles for the packet error of 4.0, 6.33 and 4.33 for the individual servers which is a big improvement.

Figure 4.12 shows similar results in terms of synchronization in relation to the frame rate. Here non-synchronized system has 90% percentiles for the frame rate error of 453.74, 581.55 and 509.58 for the individual servers and the synchronized system has 90% percentiles for the frame rate error of 4.31, 5.63 and 4.35 for the individual servers which again is a big improvement.

4.2.2 Limitations of LQ-CPU-Frequency-Synchronization

The LQ-CPU-Frequency-Synchronization method has some limitations that come with the used regulator and the way the output rate of a sever is regulated. The LQ-Regulator is designed for usage with arbitrary values (in the interval $]-\infty;\infty[$). This is not true for any practical environment. The CPU frequency cannot take arbitrary values. A server has a minimum and maximum frequency on which it can operate. The lower bound is determined by the CPU, the upper bound is mostly dictated by the cooling and power delivered to the CPU.

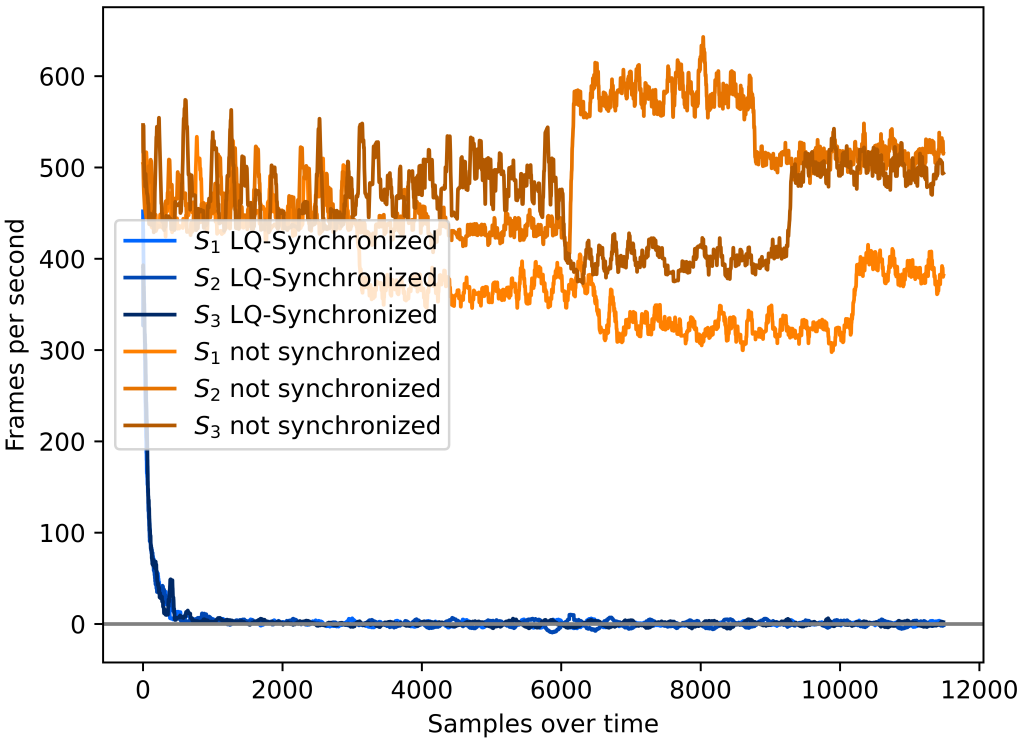

Figure 4.12: Frames per second error when synchronizing three model server

It is sufficient to clip the values provided by the LQ-Regulator in a way that the frequency stays within the acceptable range of the CPU. When implementing the approach it should be considered that a CPU can reach some frequencies only for some time (CPU clock boost) but then slows down to prevent over-heating.

Another aspect to consider is that due to the limited regulation capabilities of a server it is necessary to use servers that can reach a synchronized state at all. If there is no overlap between the possible frame rates with the CPU frequencies usable then the regulator can only speed up the slow server to the maximum and reduce the calculation power as much as possible on the faster server but they can not reach a state where they have the same frame rate.

As seen in Figure 4.13 even when disregarding network delay the servers do not get the regulation input immediately from the controller because the mobile device first needs to collect all data packets from the servers, calculate the state vector and then the controller needs to calculate the actual regulation values. In the mean time the servers may have calculated more frames. This dead-time leads to an oscillation around the set point since the regulator has only a discrete view onto the system and may regulate based on slightly out-of-date information.

4.2.3 Overview over LQ-CPU-Frequency-Synchronization

Figure 4.13 shows the three components of our system and how they operate over time. In the orange upper row are the servers that continuously calculate new frames for the mobile device to display. The red arrows are the messages that are exchanged via MQTT. After calculating a frame the server sends the data to the mobile device to display and calculate the current synchronization state which is shown in the second row in blue. The mobile device does two task asynchronously: rendering and displaying data is performed simultaneously to calculating the new state vector and sending the information packet to the controller. The lower row in green represents the controller. It receives the messages published by the mobile device and calculates the new CPU frequency changes for the servers. After calculating it publishes the information under the corresponding topics `control-packet/x`. The servers in the upper row than react to the control input and change their CPU frequency accordingly to reach the desired synchronization state. To maintain readable only one cycle of MQTT communication is shown in the image.

4.2.4 Communication for LQ-CPU-Frequency-Synchronization

The communication for the LQ-CPU-Frequency-Synchronization is done via the messaging middleware MQTT. It implements a topic based publish/subscribe system using a broker as main communication component.

In our system we have three components that need to communicate with each other. The Servers need to send their calculated data to the mobile device that will display the resulting image to the user. The mobile device needs to send the information about the synchronization state to the controller component that decides with the LQ-Regulator how the servers needs to react given the reported current state. After calculating the regulation values these need to be send from the controller to the servers so they can adapt based on the provided values by the controller.

Sending the frame information from the S_i to the mobile device is done via the topic `datapacket/i`. In order for the mobile device to receive these messages it has to subscribe to the topic `data-packet/+`. The + at the end of the topic is a single level wildcard in MQTT [Oas]. This means that the mobile device will subscribe automatically to the data packets of all servers because the plus character matches all topics on the same level (levels are separated by a / character). The mobile device still gets the actual topic that information was published to. This is important for the mobile device in order to calculate the current synchronization state because it has to map the arrival times to a specific server. Along with the calculated frame data the server also sends a sequence number so that the mobile device can calculate the packet error.

The mobile device continuously calculates the frames per second per server as well as the average of index of the last frame arrived across all servers. This average is then used to calculate the packet error per server. The mobile device then sends periodically an information packet to the controller. This information will be send via the topic `information-packet`. The controller subscribes to this topic in order to receive the updated information when the mobile device publishes new information.

The controller will then use this information and calculate via the initially calculated gain matrix K the control input for each server. This is done by calculating $u = -Kx$. K is calculated using the A, B, Q and R matrices from Section 4.2. x is created by the controller as a vector from the

values provided by the mobile device as described in Section 4.2. The result of the calculation $-Kx$ is a vector that contains the information which server needs to increase their CPU clock frequency and which server needs to decrease their CPU clock frequency in MHz. The controller publishes the delta of the clock frequency that S_i needs to speed up or slow down on the topic `control-packet/i`.

The servers subscribe only to the topic that concerns their control input (e.g. S_3 only subscribes to `control-packet/3`). When receiving a control packet it will increase or decrease it's CPU clock frequency according to the information contained in the control packet provided by the controller.

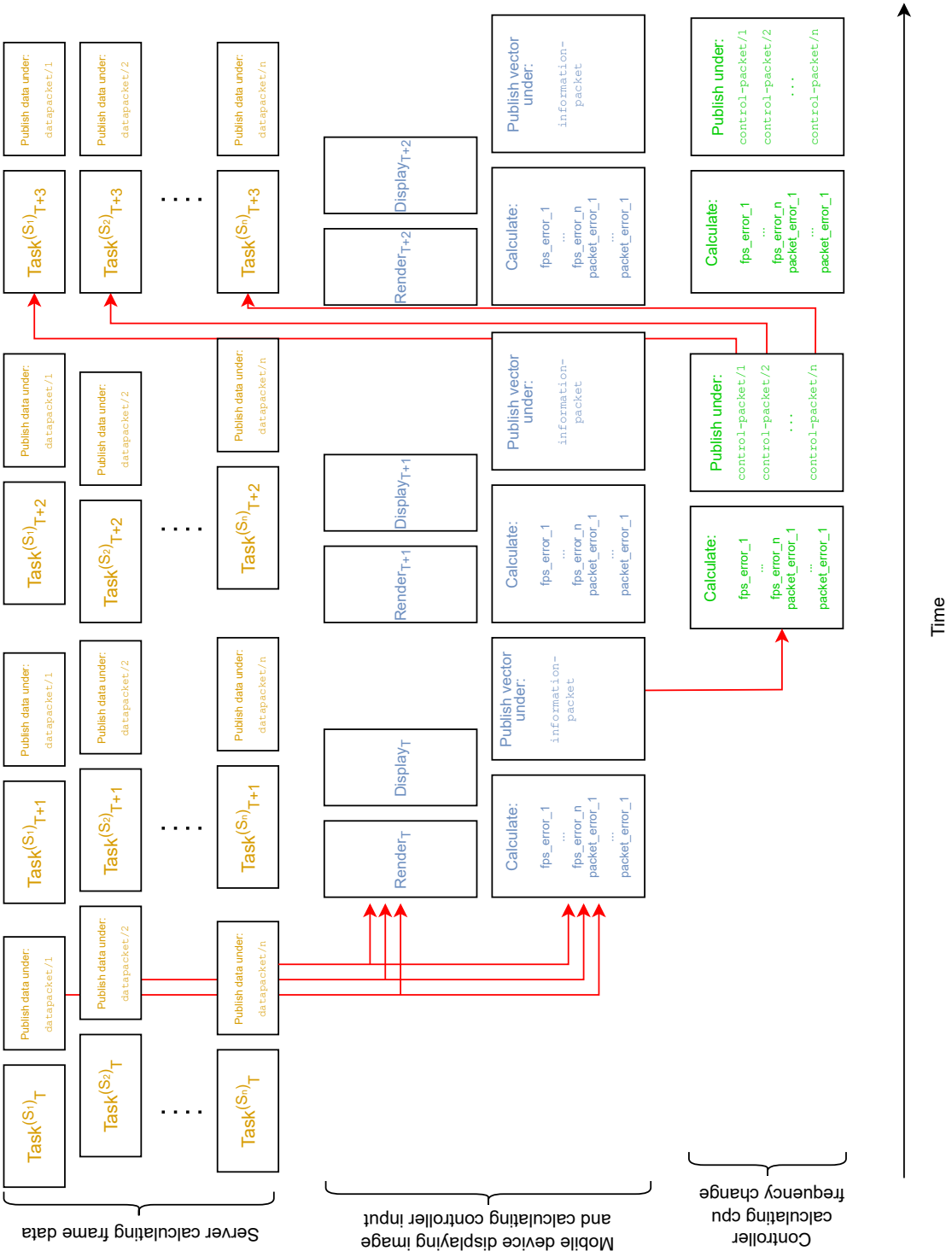

Figure 4.13: Chronological sequence of LQ-CPU-Frequency-Synchronization

5 Implementation

To proof the validity of our concepts and to later evaluate the performance of our approach, we present a proof-of-concept implementation of our system in this chapter.

The implementation consists of four main parts.

- The interface that allows for the control of the CPU frequency of the servers.
- The controller that takes the input of the mobile device and calculates the control instructions for the server.
- Load generators that simulate the typical load generated by neural networks as would be used by a distributed simulation system.
- An MQTT-Broker that is used for the communication.

A sequence diagram for the proof of concept implementation is given on the right side of Figure 5.1. The controller is a python program that periodically requests the current state of the server. This is done via one MQTT message that is received by all computing servers (first arrow in Figure 5.1). The frame producing servers also use a python program. This program receives the messages from the MQTT broker and sends the current state including the current frame rate and the sequence number of the last calculated frame back to the controller via MQTT, this will be explained later in further detail. This communication is represented by the three arrows labeled ”Response S_1” to ”Response S_n” in Figure 5.1. After receiving all responses of the servers the controller creates the state vector as described in Section 4.2.1 and calculates the regulation values. These are then send to the servers which is represented by the last arrow. The MQTT broker is running on the same server as the controller.

5.1 Simplifications of the Proof of Concept

To eliminate errors that are not related to our approach, we decided to test the system without a mobile device. The performance could be effected for example by a slow network card, a busy operating system (lots of background tasks unrelated to the simulation) or a bad wireless connection. The difference between the approach and the proof of concept is shown in Figure 5.1. The left side shows the approach and the right side the simplified proof of concept version. The servers are discarding the calculated simulation result and do not send it to the broker. To compensate for this constrain, we added a delay to the system to simulate the time it takes to send data from the servers to the mobile device and for the mobile device to the controller.

Listing 5.1 Function for calculating shifted uniform distribution

```
from random import random
def get_network_delay(expected_value, spread):
random_part = random() - 0.5
return expected_value + random_part * spread
```

In our approach the mobile device would calculate the state vector, but since we do not have a mobile device in our proof of concept implementation we decided to relocate the calculation to the controller itself. Since the calculation of the state vector and its needed values is not costly, it does not matter for the evaluation of the synchronization mechanism if it is calculated on the mobile device or by the controller.

The regulator would normally get a message including all information from a specific point in time from the mobile device. To simulate this, instead of the mobile device sending the data to the controller, the controller requests the data from the servers which respond with their current frame rate and the index of the last produced frame (latest sequence number) as seen in Figure 5.1. This gives the controller a snapshot in time for the needed values of the state vector.

To get an understanding what effect a network delay might have on our system, we added a random delay to the messages that are exchanged by the controller. This simulated network delay can be enabled or disabled. For the probability distribution, we chose a shifted uniform distribution. When enabled, we chose a random network delay of 250 to 350 milliseconds with the following code in Listing 5.1 `get_network_delay(0.2,0.1)` (a 100-millisecond constant delay is added to this). These are quite pessimistic assumptions, the network delay of a local area network is in the range of tens to a few hundred micro-seconds, even in wide-area networks the network delay rarely exceeds a few hundred milliseconds.

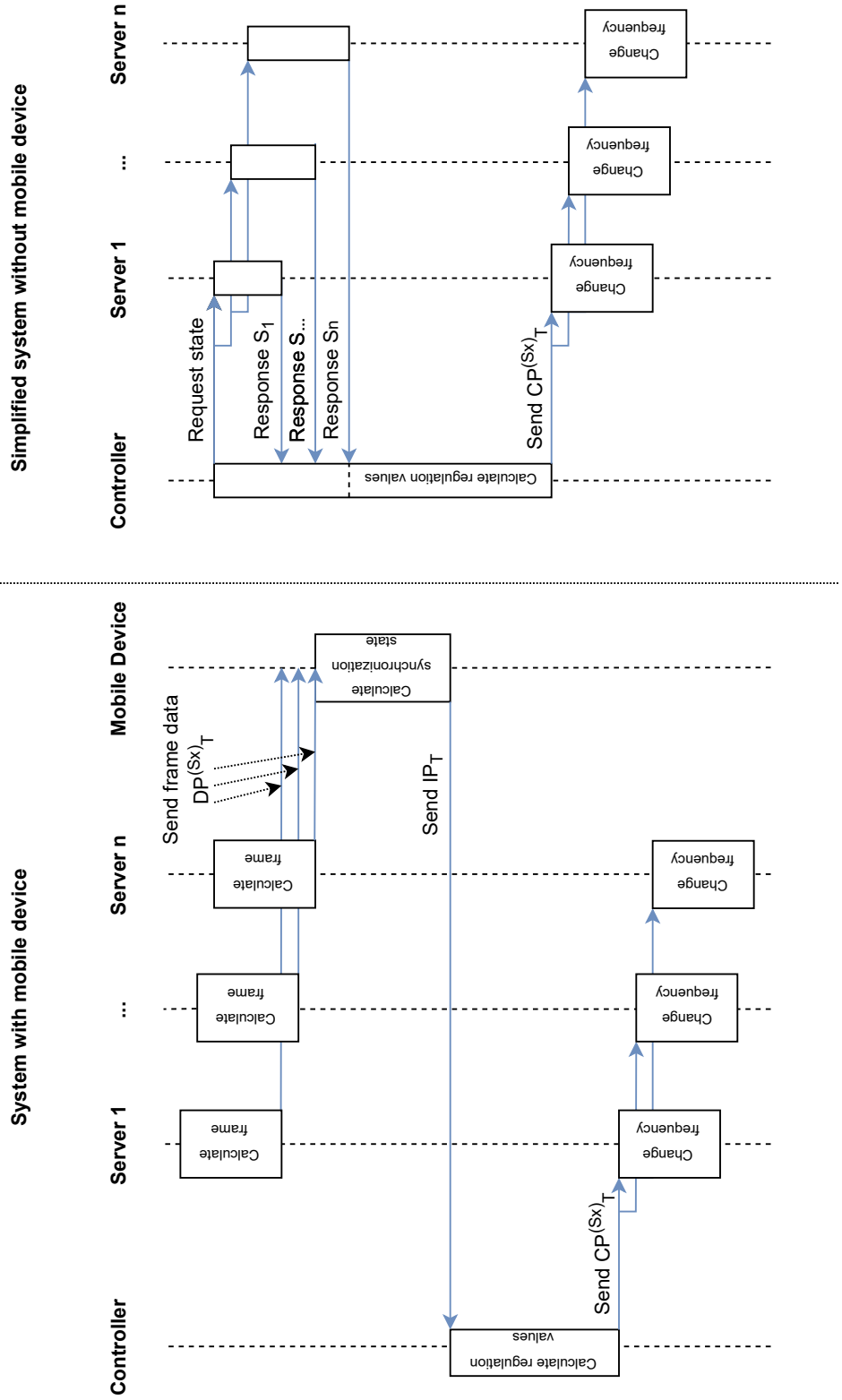

Figure 5.1: Approach compared to simplified proof of concept

5.2 CPU-Frequency Interface

The software that regulates the CPU frequency is called CPU governor. Each governor has its own rules based on which it will regulate the CPU frequency. There are several available governors depending on the system and the activated CPU drivers.

These are some examples for CPU governors:

Performance tries to set the CPU frequency to the maximum possible frequency within the limits of the CPU.

Powersafe tries to set the CPU frequency to the lowest possible frequency within the limits of the CPU.

Userspace Allows any user that is running with the UID ”root” to set a specific CPU frequency.

Ondemand sets the CPU frequency according to the current system load. If there is a high system load then the CPU frequency is increased, if the system load is low then the CPU frequency is reduced.

Since we want to regulate the cpu frequency through the implemented controller we use the `userspace` governor. This governor does change the cpu frequency based on a file from the sys pseudo file system. We implemented our proof of concept system on a Debian Bullseye system. Here we have two options to set the CPU frequency. The first option is to write the CPU frequency to the pseudo file `/sys/devices/system/cpu/cpu0/cpufreq/scaling_set_speed`. This can be a bit inconvenient, which is why the package `cpufreq` was developed. This package for Debian simplifies the process of reading the current CPU state and setting it if possible. With the command `cpufreq-info` the current governor can be checked as well as looking up the available ones. A prerequisite of our approach is that the userspace governor is available. If the command does not list the userspace governor, it is possible that it can be activated as a Linux kernel module. It is also possible that the userspace governor is not supported, in this case the approach is not implementable on the specific system unless another way to set the CPU frequency exists.

To make the process of setting the CPU frequency easier, we used the Debian package `cpufrequtils` [Han]. In our case, the output of the command `cpufreq-info` is shown in Listing 5.2.

The line *available cpufreq governors:* tells us that the userspace governor is available. To use it, we also need to set it as active governor (which was already done before the output of Listing 5.2). For this, the `cpufreq-set` command can be used. With `cpufreq-set -c 0 -g userspace` the governor for core zero is set to the userspace governor. This needs to be done to all available cores. After setting the userspace governor the output of `cpufreq-info` should state for all cores (there may be systems that can handle different clock speeds for the individual cores, but our test system did not support this) at the end `The governor 'userspace' may decide which speed to use within this range.`

After setting the governor for all CPU cores, the CPU frequency can be set with the command `cpufreq-set -f 2.0 ghz -c0`. This will set the CPU frequency of core zero to 2.0 GHz. To specify the core, the last value needs to be adjusted to the core index that should be changed. In our tests, we noted that it is beneficial to set the minimum and maximum speeds as well. We set the minimum and maximum CPU frequencies to the same frequency as the target frequency with the commands

Listing 5.2 Beginning of the output of `cpufreq-info` on the test servers

```
    ~$ cpufreq-info
    cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
    Report errors and bugs to cpufreq@vger.kernel.org, please.
    analyzing CPU 0:
    driver: intel_cpufreq
    CPUs which run at the same hardware frequency: 0
    CPUs which need to have their frequency coordinated by software: 0
    maximum transition latency: 20.0 us.
    hardware limits: 1.60 GHz - 3.80 GHz
    available cpufreq governors: userspace, conservative, powersave, ondemand, performance,
schedutil
    current policy: frequency should be within 2.20 GHz and 2.20 GHz.
    The governor "userspace" may decide which speed to use
    within this range.
    current CPU frequency is 2.20 GHz.

    [...]
```

`cpufreq-set --min 2.0GHZ -c0` and `cpufreq-set --max 2.0GHZ -c0`. The maximum and minimum values that the regulator is able to set can be seen in the output of Listing 5.2. The line *hardware limits:* tells the upper and lower limit of the CPU frequency for the used hardware.

Note that the `cpufrequtils` commands used above need to be executed with root rights. [BGWK13]

5.3 LQ-Regulator

For the LQ-Regulator we used the python library `control` which provides the needed functionality to calculate the gain matrix from the matrices A, B, Q and R. To calculate K the library provides the function `control.lqr(A, B, Q, R)` where the parameters are matrices defined by numpy arrays.

Listing 5.3 Calculating gain matrix in python

```
A = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
B = np.array([[1, 0], [0, 1], [1, 0], [0, 1]])
Q = np.array([[400, 0, 0, 0], [0, 400, 0, 0], [0, 0, 1000, 0], [0, 0, 0, 1000]])
R = np.array([[1000, 0], [0, 1000]])

K, S, E = control.lqr(A, B, Q, R)
```

As seen in Listing 5.3 a matrix is defined by a list of rows which are also lists. The lqr method of the `control` library [pyt] not only provides the gain matrix K but also the solution to the Riccati equation S and the eigenvalues E. For our approach, we only need the gain matrix.

While calculating the regulation values of the servers, the controller repeats the following instructions: First it sends out the request to the servers to report the current frame rate and the index of the last produced frame (sequence number). Then it calculates the state vector for the LQ-Regulator. Next, it gets the new control instructions for the servers by calculating $-Kx$. Then it sends the instructions to the individual servers via MQTT.

The gain matrix K is only calculated once at the beginning of the program because it is not dependent on the state vector. It can therefore be used for the complete runtime of the program or even be defined offline a priori and passed to the program as static parameter if the matrices A, B, Q and R did not change.

5.4 CPU Load Simulation

The calculation of a neural network in regards to the CPU load is no different then any computational intensive calculation that is performed on the CPU. Because of this we chose, for the calculation that is synchronized by the regulator, to use a replaceable black box calculation which we will call load generator. This load generator should be able to be adjusted dynamically to simulate a range of CPU loads.

Our load generator is a function that calculates pi up to a certain decimal point. To vary the CPU load the amount of decimal points calculated can be adjusted dynamically. Increasing the calculated decimal points that are calculated will increase the CPU load and the other way round. To do this, we use an implementation of the Chudnovsky algorithm. As mentioned in Section 4.1 we only calculate pi to the $\sqrt{x}$ decimal point which is why we take the square root of the input value and also ensure that a positive amount of decimal points is calculated by constraining the input with the `max([...])` function to values ≥ 1.

For a real application with neural networks for the calculation of a distributed simulation this black box load generator can easily be replaced by swapping the load generator function by the calculation of the neural network. In other words a neural network for the calculation of a concrete pervasive simulation is just an instance of a load generator. The use of a neural network would actually add constrains because we can not easily adjust different loads.

Our evaluation is therefore not only valid for the calculation of neural networks but for any calculation that can be used as load generator.

5.5 Communication

The communication with MQTT consists of two components, the broker and the MQTT clients. For the MQTT broker, we decided to use the Eclipse Mosquitto implementation. This is an open-source message broker that implements the MQTT versions 5.0, 3.1.1 and 3.1 [Fou]. For our purpose, the MQTT version does not matter because we only need basic functionality that is supported by either of the supported versions.

For the MQTT clients, we used the Python library `paho-mqtt`. This library supports the same versions of the MQTT standard that the Mosquitto broker supports.

Listing 5.4 Modified Chudnovsky algorithm in python [w3r]

```
def compute_pi(self, n):
  n = int(math.sqrt(max(n, 1)))
  decimal.getcontext().prec = n + 1
  C = 426880 * decimal.Decimal(10005).sqrt()
  K = 6
  M = 1
  X = 1
  L = 13591409
  S = L
  for i in range(1, n):
    M = M * ((1728*i*i*i)-(2592*i*i)+(1104*i)-120)/(i*i*i)
    L += 545140134
    X *= -262537412640768000
    M, L = decimal.Decimal(M), decimal.Decimal(L)
    S += decimal.Decimal((M*L) / X)
  pi = C / S
  return pi
```

For all messages, we use the quality of service of zero this means that the messages are sent with best effort to and from the broker but the delivery of the messages is not guaranteed under for example network failure. The first reason for this is that messages transmitted with QoS zero can be sent faster than with a higher QoS of one or two. This is because the higher levels of QoS need additional messages to ensure the delivery of the message. It is not crucial for our system that all messages arrive at their destination. If a message gets lost, then the synchronization information for a few frames get lost. Because we simplified the system for the proof of concept implementation, we do not send the actual calculated frame data. It can be considered using at least QoS of one for these messages, since skipping a frame may be visible to the user while single skipped synchronization messages are less noticeable.

The second reason is that the frequently sent synchronization packets make the lost packets that would need to be resend obsolete. If a packet is re-transmitted after a new packet was calculated and sent it we would update the cpu frequency based of outdated information which is unnecessary at best and counter productive at worst.

In summary, we use QoS zero because it would make sending the messages slower and we have no benefit by increasing the QoS.

6 Evaluation

In this chapter we validate our concepts and evaluate their performance using the previously presented proof-of-concept implementation.

6.1 Evaluation Setup and Scenarios

For our tests, we if not stated otherwise used one dedicated server for both the MQTT broker and the controller that calculates the CPU speed changes. This is needed because the controller continuously calculates regulation values, which takes up computational power. If the controller component would be on the same system as the program that calculates the frames for the mobile device, it would be affected by its own regulation.

This could be problematic if the regulator slows down the server it is running on because the regulator would calculate the regulation values slower and therefore the servers would get slower feedback on how they should behave. However the calculation of the controller is just one matrix multiplication per control message. The calculation time per control message is therefore small compared to the waiting time for the synchronization state messages. We also tested a version where the controller runs on the same server as a frame calculating program and found that with our system the regulator was still able to control the servers as we expected, but we are not focusing on this scenario because of the light overhead of the calculation.

We tested two server setups.

In the first *Homogeneous Server Setup* we used homogeneous hardware for the frame calculating servers. The servers used were two Intel Xeon x86_64 servers.

The second *Heterogeneous Server Setup* used heterogeneous hardware to see if the approach is also working in such environments. The used servers in this setup are two Intel Xeon x86_64 servers and one Raspberry Pi 4 compute module. The calculation of the Raspberry Pi that was synchronized was less complex compared to the calculation on the Intel servers. We explain why this is the case in Section 6.4.

Each setup was tested in three scenarios:

Zero-Delay Scenario here we tested how the system is behaving if there is no simulated network delay.

Constant Delay scenario where we added a constant network delay of 100 ms to the exchanged messages. Similar to what could be expected in a lightly loaded network scenario with good wireless network connectivity on the last wireless link (zero bit errors). Because our approach is intended to be used in an edge cloud, this network delay should be sufficient to test the effectiveness of our approach since the delay would in most cases not be higher than this.

Random Delay scenario where we added a random network delay of 250 ms to 350 ms (uniformly distributed) to the exchanged messages. Similar to what could be expected either in a network with significant load (queuing delay) and/or wireless link with significant bit error rate leading to re-transmissions which show as random delay. The network delay will never be zero, which is why we shifted the uniform distribution in order to have a minimum delay. The network delay is uniformly distributed between 250 and 350 milliseconds.

These are both pessimistic network delay assumptions to test for the worst case.

As performance metrics, we used the frame rate error, packet error and the frame rate variance. The metrics are defined as follows:

Frame Rate Error: $\texttt{frame\_rate\_error[i]} = \texttt{frame\_rate[i]} - avg(\texttt{latest\_packet})$

Packet Error: $\texttt{packet\_error[i]} = \texttt{latest\_packet[i]} - 100$ (100 frames per second was the set point)

Frame Rate Variance: $s^2 = \frac{1}{n}\sum_{i=1}^{n}(\texttt{calculation\_time[i]} - avg(\texttt{calculation\_time}))^2$

In each scenario we tested if the servers could reach an initial synchronization state after starting the test, whether the regulator could hold this state and also how the regulator reacts to dynamically changing the complexity (disturbance) of the calculation on different servers.

The runtime of each experiment was six minutes, for which we recorded the frame rate error and packet error over time.

6.2 Homogeneous Server Setup

This section describes the *Homogeneous Server Setup* defined earlier with its three scenarios.

6.2.1 Zero-Delay Scenario

In Figure 6.1 the frame rate in frames per second is shown on the y-axis. In Figure 6.2 on the y-axis the packet error is shown. Each sample on the x-axis is taken before the regulator calculated the control information for both images.

We also provide for each scenario a table showing the 90% percentiles for the frame rate error, packet error and frame rate variance.

Figure 6.1 and Figure 6.2 show the behavior of the servers in the Zero-Delay Scenario by showing how the frame rate error and packet error evolved over time. The images show that the frame rate stays around the set point of 100 frames per second. There are some points at which the frame rate deviates from the set point. At these points, our test program simulated a change in complexity of the calculation. The controller was able to adapt to these changes because the frame rate returned to the set point. Figure 6.2 also shows the changes of the complexity match the points in time when Figure 6.1 deviates from the set point, which is to be expected. Table 6.1 shows the 90% percentiles for the frame rate error, packet error and frame rate variance for the images Figure 6.1 and Figure 6.2. With values of less than 2.9, 3.1 and 1.5 for the frame rate error, packet error

and frame rate variance respectively, we showed that the LQ-CPU-Frequency-Synchronization is able to keep the frame rate at the set point and adapt to dynamic changes in the complexity in an environment with two servers when disregarding the network delay.

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	2.695	3.000	1.397
S_2	2.852	3.000	1.463

Table 6.1: 90% percentile for two servers without network delay

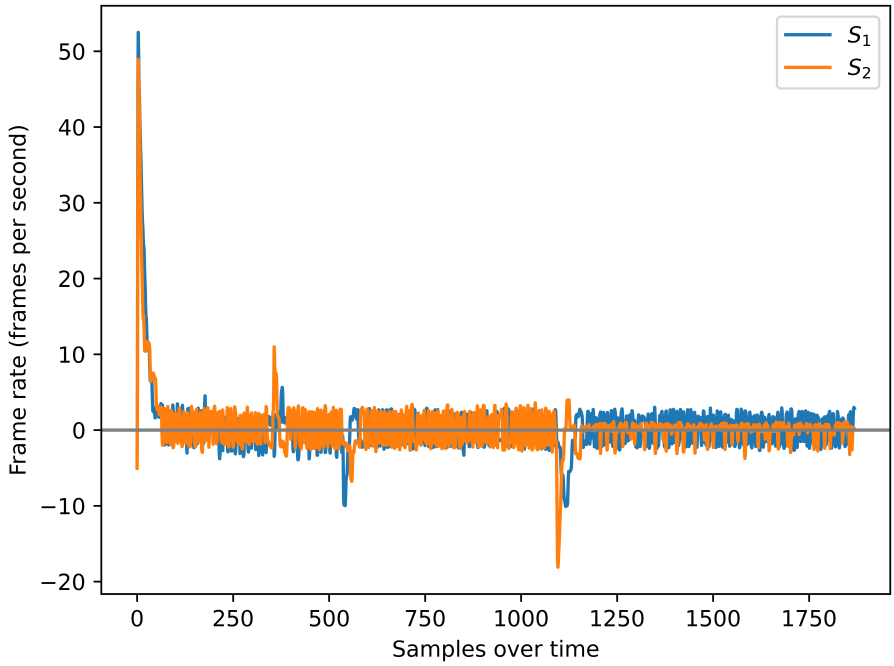

Figure 6.1: Frame rate error of two servers without network delay

6.2.2 Constant Delay Scenario

Because a system without network delay is not a realistic one, we also tested our implementation against a constant network delay of 100 milliseconds.

Figure 6.3 and Figure 6.4 show the resulting frame rate errors and packet errors. Again the LQ-CPU-Frequency-Synchronization is able to correct all deviations from the set point caused by either the initial synchronization when starting the regulator as well as adapting to the changing complexity at the x-axis points around sample 200, 300 and 600. Comparing Figure 6.4 and Figure 6.2 it can be observed that the server in Figure 6.4 oscillates longer. This can be explained by the regulator getting less frequent updates and therefore being less often able to update the CPU frequency and therefore adapting slower.

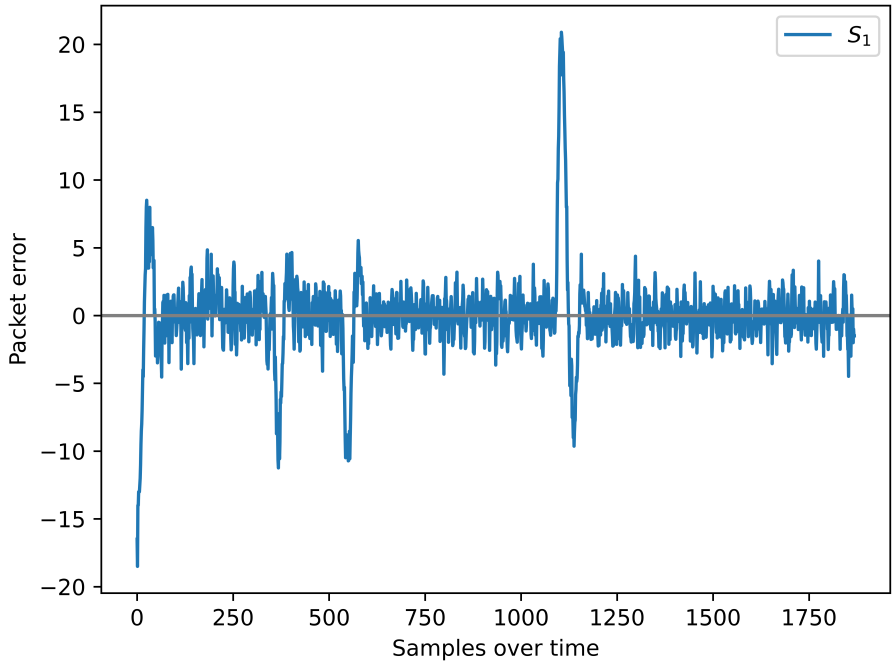

Figure 6.2: Packet error of two servers without network delay

Table 6.5 confirms this, with the 90% lows of the frame rate error being about 0.7 higher. The changes in the packet error are more noticeable, being seven more in the scenario with simulated constant network delay compared to without network delay. The variance however seems to be not affected by the simulated network delay.

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	3.419	10.000	1.460
S_2	3.611	10.000	1.467

Table 6.2: 90% percentile for two servers with constant network delay (100 ms)

6.2.3 Random Delay Scenario

The last test for the *Homogeneous Server Setup* was to change the network delay to have a shifted uniform distribution because the network delay in a real application will not stay constant.

Figure 6.5 and Figure 6.6 show similar behavior as from going from no network delay to a constant network delay. The LQ-Regulator is still able to synchronize the system, but it takes longer to do so. This is again confirmed by the 90% percentiles. Table 6.3 shows that the frame rate values are nearly doubled to around 6.7 to 7.2 the packet error increased by about 76% to 17.6. In this test, the variance even decreased compared to the two tests before by around 45%.

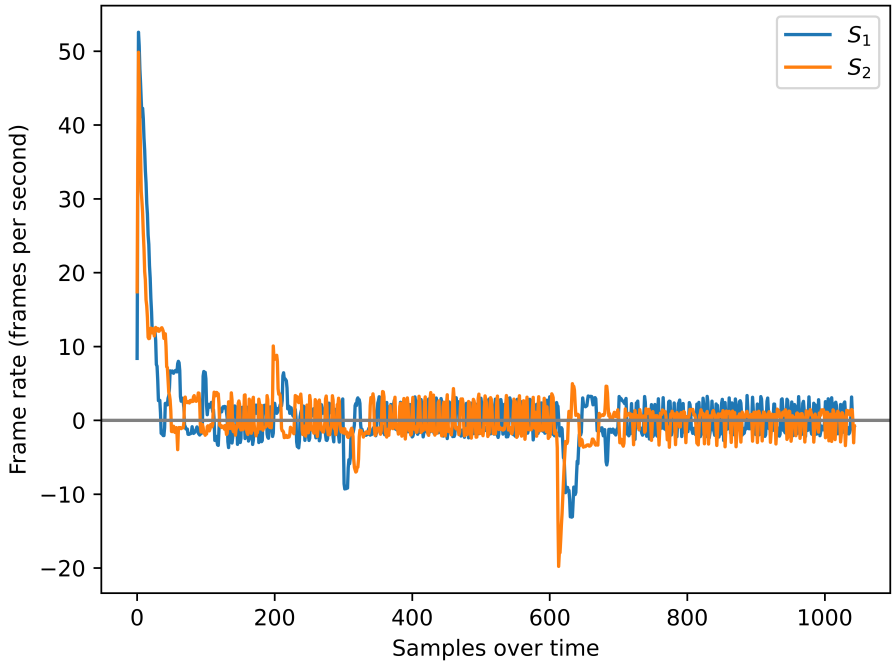

Figure 6.3: Frame rate error of two servers with 100 ms network delay

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	7.210	17.599	0.766
S_2	6.789	17.599	0.789

Table 6.3: 90% percentile for two servers with random network delay (250-350 ms)

Figure 6.7 shows a summary of the observations from above. Bars of the same color represent the different servers, going left to right S_1 and S_2, while the colors blue, yellow and green going left to right represent the individual tests (without network delay, constant network delay, random network delay). This image clearly shows the increased frame rate error and packet error with increased network delay and the not strongly affected variance of the frame rate.

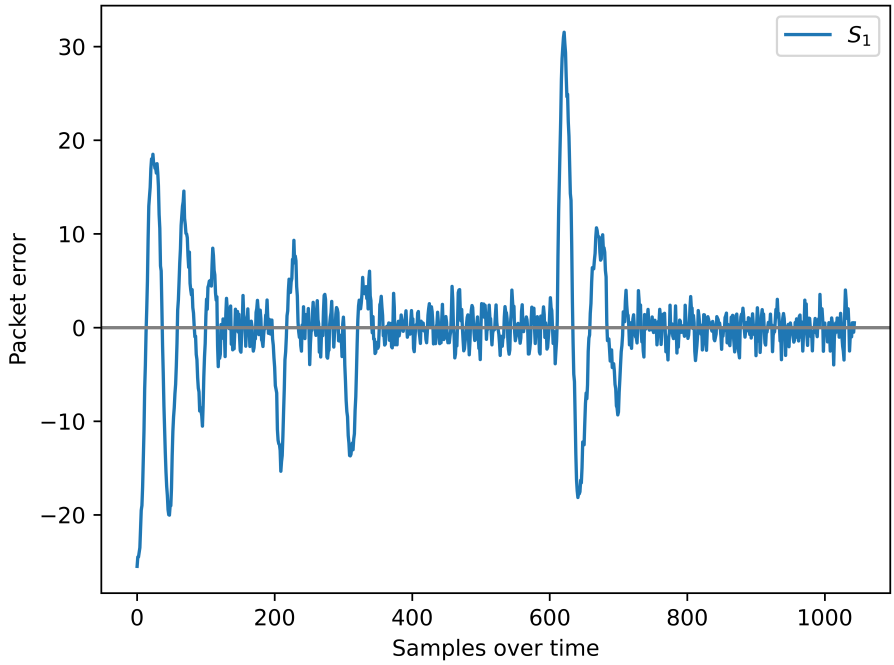

Figure 6.4: Packet error of two servers with 100 ms network delay

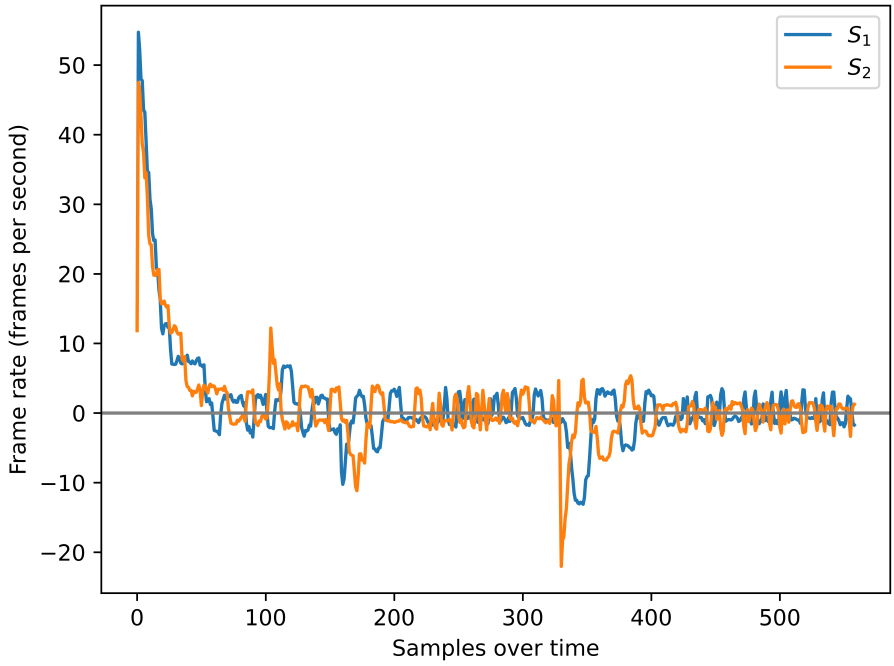

Figure 6.5: Frame rate error of two servers with 250-350 ms random network delay

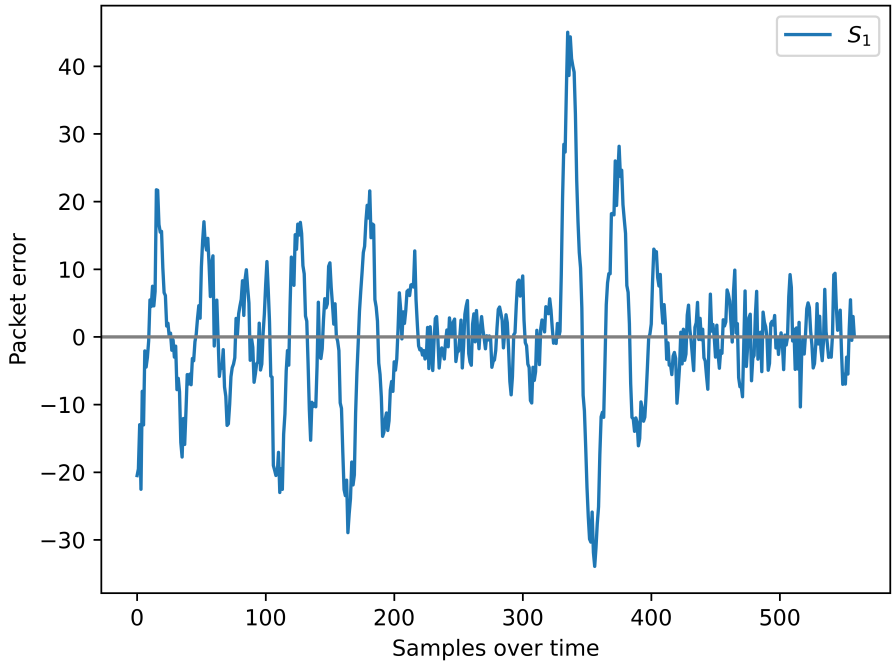

Figure 6.6: Packet error of two servers with 250-350 ms random network delay

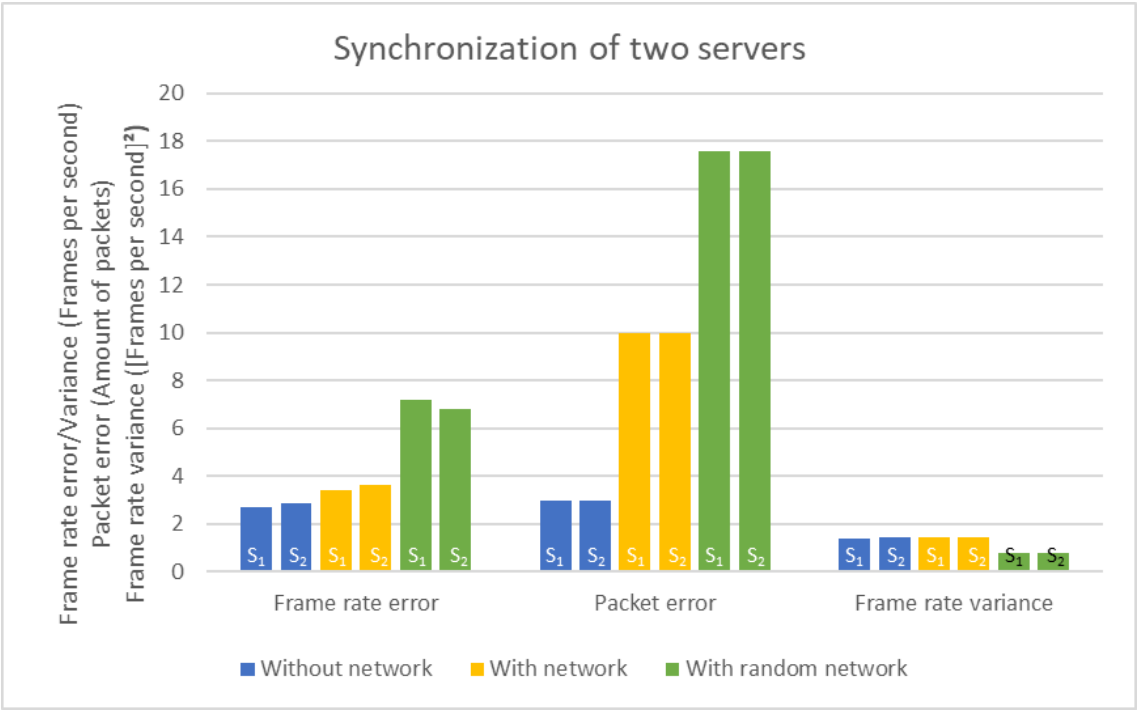

Figure 6.7: Evaluation summary of two servers

6.3 Heterogeneous Server Setup

The second setup that we evaluated consists of two homogeneous Intel Xeon x86_64 servers and a low-power ARM server (Raspberry Pi 4 compute module) that calculate the frames for the mobile device and an Intel Xeon x86_64 as server that hosts the MQTT broker and the controller.

6.3.1 Zero-Delay Scenario

Figure 6.8 and Figure 6.2 show that the LQ-CPU-Frequency-Synchronization method is also working in the Zero-Delay Scenario with the Heterogeneous Server Setup. In the following images, S_1 and S_2 are representing the faster Xeon servers and S_3 represents the Raspberry Pi 4. It can be observed that the frame rate of S_3 is closer to the set point than S_1 and S_2 which is confirmed by the 90% percentiles showed in Table 6.4. S_1 and S_2 also appear to be regulated to be slightly under the set point, which would explain the higher 90% percentiles. This means the Raspberry Pi generally produces slightly more frames than the Xeon servers. This should intuitively mean that the Raspberry Pi is ahead of the other servers. However, the image shows that the Raspberry Pi is slightly behind the other servers because the green line is generally lower than the orange and blue line. The frame rate numbers only accounts for the calculation of the frame, which means the Raspberry Pi is calculating the frames faster than the Xeon servers but loses time due to other factors. These factors could be for example less efficient network communication with longer internal delays. Table 6.4 shows that the variance is also about double the variance of both Xeon servers.

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	6.475	9.666	1.214
S_2	7.149	11.000	1.439
S_3	4.540	17.433	2.657

Table 6.4: 90% percentile for three servers without network delay

6.3.2 Constant Delay Scenario

The next test was again adding a constant network delay of 100 ms. Here we can make the same observations as without network delay. Figure 6.10 and Figure 6.11 show that the Raspberry Pi is again regulated in a way that it produces the frames faster but lacks behind in regard of the packet error. The overall frame rate error is however balanced, as seen in the 90% percentiles in Table 6.5. This also shows that the packet error of the Xeon servers are nearly the same (only 0.3 difference) but the Raspberry Pis packet error is noticeable higher than both of the Xeon servers. This is even more noticeable for the variance, where the Raspberry Pi has nearly 4.6 times the variance than the Xeon servers.

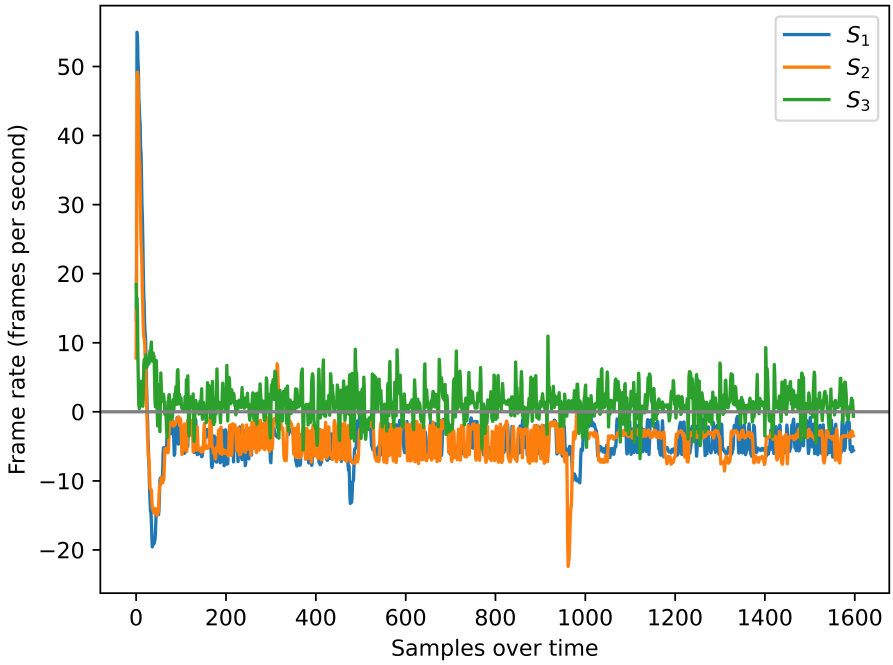

Figure 6.8: Frame rate error of three servers without network delay

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	6.206	13.000	1.248
S_2	6.771	13.333	1.282
S_3	5.881	15.333	5.697

Table 6.5: 90% percentile for three servers with constant network delay (100 ms)

6.3.3 Random Delay Scenario

The last test included the random network delay of 250-350 ms. Here, the images Figure 6.12 and Figure 6.13 show different results than in the tests before. Here it can be observed that the packet error of Raspberry Pi is now lower than the ones of the Xeon servers. The Raspberry Pi also has no longer a higher frame rate error than the Xeon servers. The variance of the Xeon servers stays low, but the variance of the Raspberry Pi increases significantly. In this test, it is nearly 18 times higher than the variance of the Xeon servers.

Figure 6.14 summarizes the tests of the heterogeneous hardware. The three bars going left to right in the same color are the servers S_1, S_2 and S_3 respectively. The frame rate error does not seem to be affected by the addition of a constant or random network delay. The packet error of the faster Xeon servers is not significantly affected by adding a constant network delay, but increases significantly when adding the random network delay. The packet error of the Raspberry Pi is not as much affected by the increased simulated network delay. It is not obvious to us why this is the case

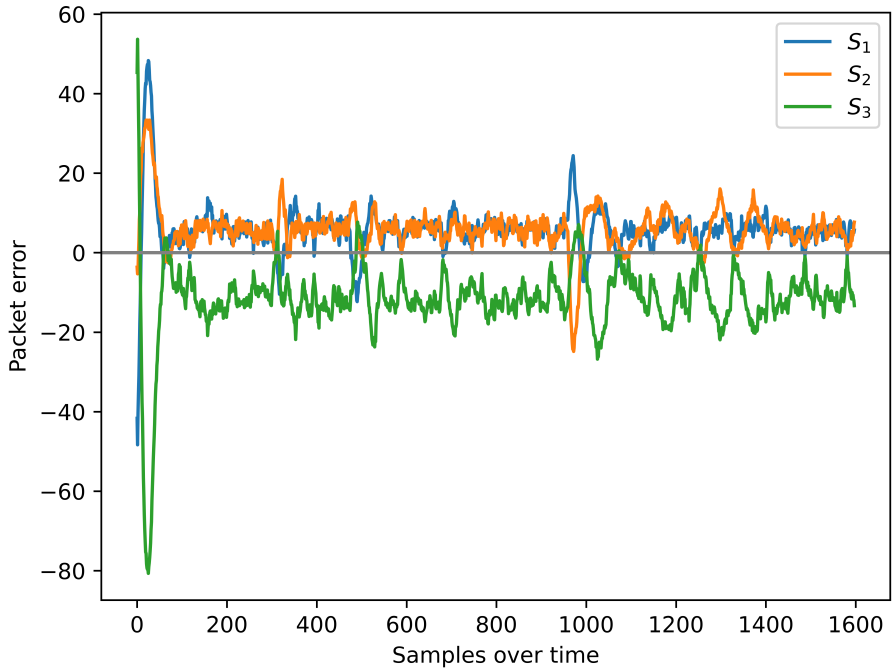

Figure 6.9: Packet error of three servers without network delay

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	6.641	29.966	0.776
S_2	6.690	28.633	0.851
S_3	6.920	22.600	14.380

Table 6.6: 90% percentile for three servers with random network delay (250-350 ms)

which is why finding the reason for this behavior would need further analysis. The variance of the Xeon servers is not significantly affected by an increased network delay. However, the variance of the Raspberry Pi is increased hugely with increased network delay, from about three to around 14.

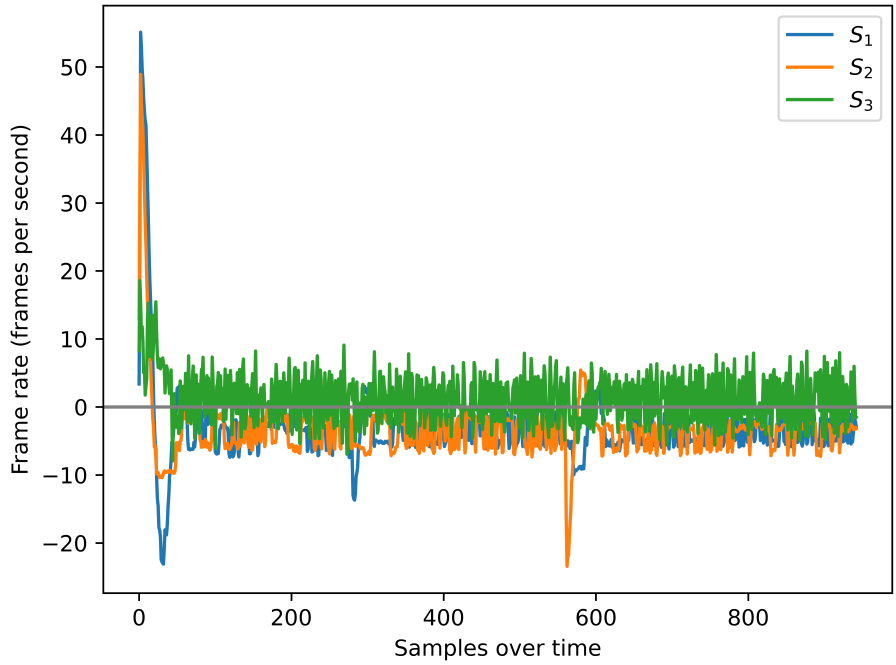

Figure 6.10: Frame rate error of three servers with 100 ms network delay

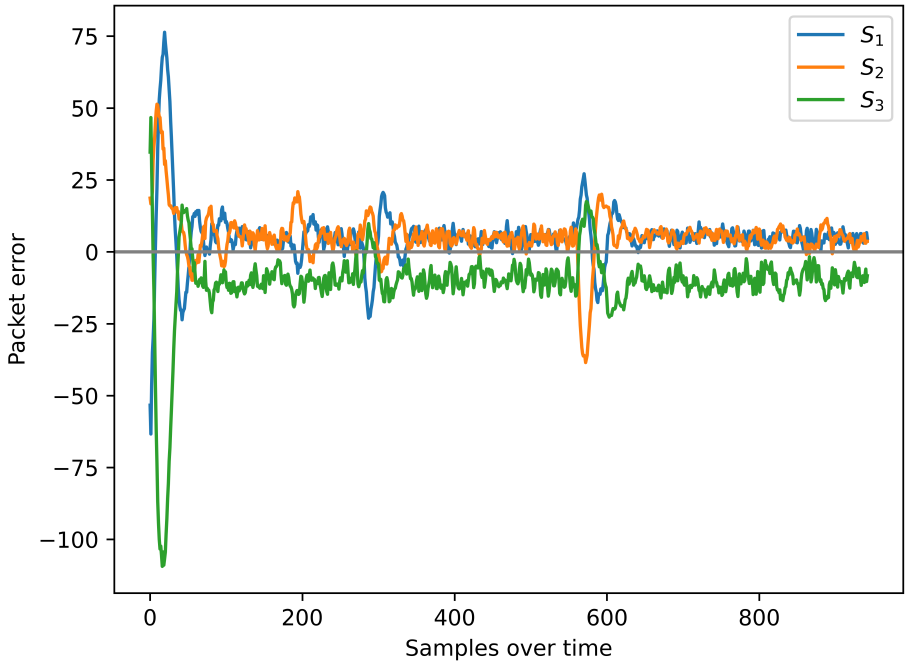

Figure 6.11: Packet error of three servers with 100 ms network delay

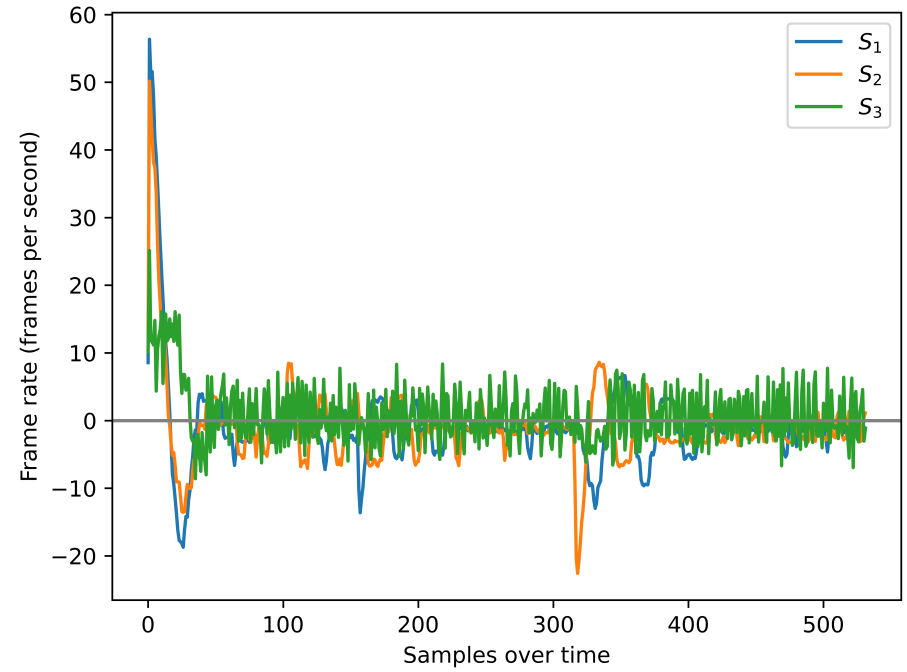

Figure 6.12: Frame rate error of three servers with 250-350 ms random network delay

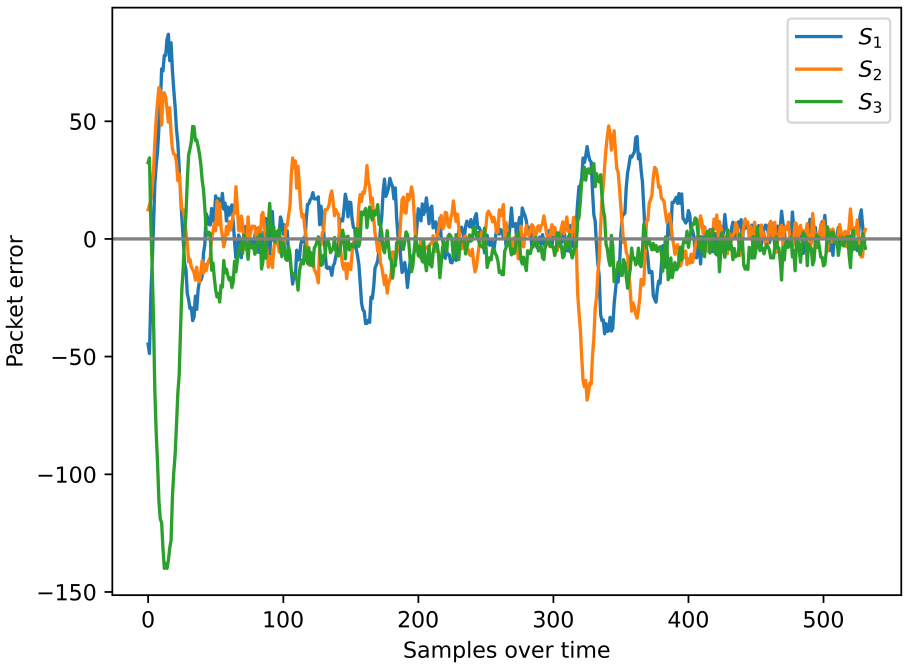

Figure 6.13: Packet error of three servers with 250-350 ms random network delay

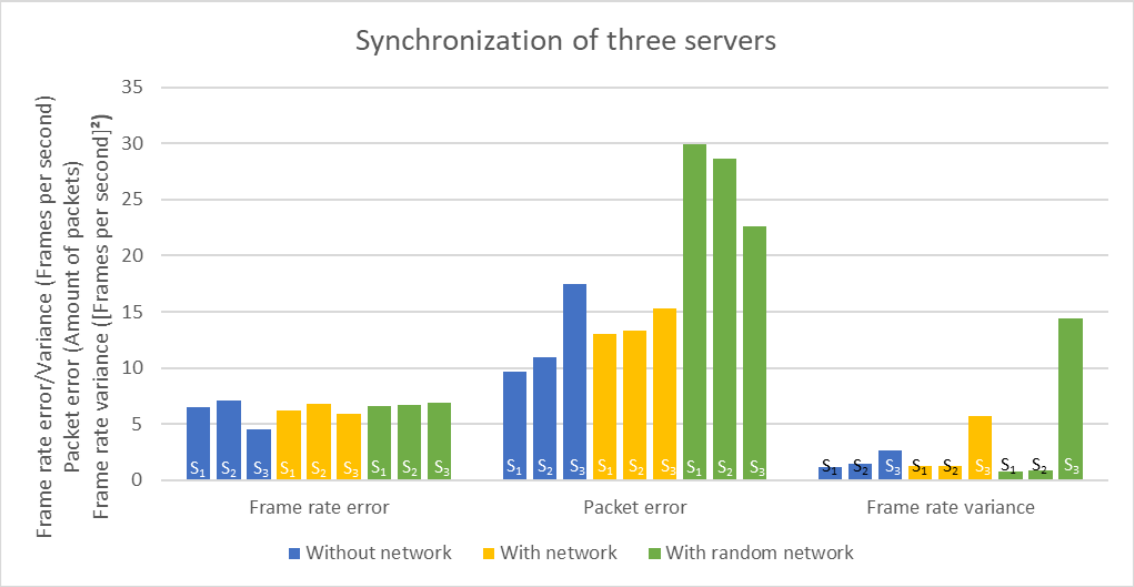

Figure 6.14: Evaluation summary of three servers

6.4 Limitations

After we have presented the results of our approach previously, we next want to discuss some limitations and open questions to show possible future research directions

The approach comes with the limitation that the regulator cannot regulate the CPU frequency of the servers to arbitrary values. This means that the calculation of the server can only be so complex that the regulator does not need to regulate the system above the maximum or under the minimum acceptable CPU frequency for the system.

Figure 6.15 shows what happens if one of the servers cannot be regulated to the desired frame rate. S_1 and S_2 could be regulated in a way that the frame rate target can be hit, but the calculation on S_3 is too complex, meaning the CPU frequency cannot be regulated high enough to hit the frame rate target.

What happens then is that the regulator regulates the slow server to the maximum possible CPU frequency to get as close as possible to the target frame rate. The other servers will slow down as much as possible in order to keep the packet error as low as possible. The jumps in frame rate of S_1 and S_2 are the simulated changes in complexity. The frame rate error changes because the CPU frequency always gets regulated to the lowest possible frequency, but the calculation is less or more complex.

This is why the tests before did not use the same complexity for the calculation. The Raspberry Pi calculated a less complex calculation (fewer pi decimals) than the Intel Xeon servers.

Server	Frame rate error	Packet error	Frame Rate Variance
S_1	37.178	7877.666	0.022
S_2	40.555	9464.066	0.031
S_3	71.897	17341.733	0.017

Table 6.7: 90% percentile for three servers that can not be synchronized

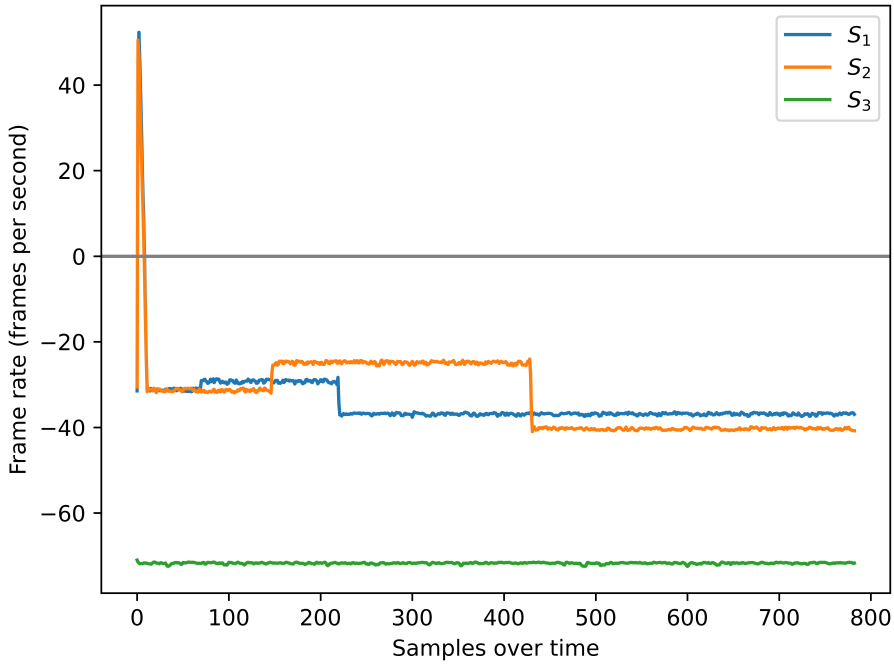

Figure 6.15: Frame rate error of three servers that cannot be synchronized

7 Summary and Future Work

In this work, we designed concepts to synchronize the output from different servers participating in a distributed mobile simulation or distributed calculation in general. The goal was to adapt the speed of calculations of the servers such that the output from multiple servers arrive at the same time and with the same rate at the mobile device, which is then presenting the results to the user.

The designed LQ-CPU-Frequency-Synchronization method is used to synchronize a concurrently running calculation on multiple servers. Each server is supposed to calculate a part of the simulation with neural networks. The simulation results are displayed in real time to a user who wears an AR-Device. Because of this, the calculation needs to be fast enough to display it without causing motion sickness and also be synchronized to reduce the necessity of buffering on the mobile device. Buffering would lead to a delay between the motion of a user and displaying the calculated simulation which leads to a decoupled movement of the user to the displayed image which can also cause motion sickness.

To synchronize the servers, the mobile device calculates a state of synchronization. This synchronization state is based on the current output rate of the servers and also a metric that shows if a server is lacking behind with producing frames or if it is ahead.

This information is sent to a controller via a topic based publish/subscribe system. The feedback control loop then calculates based on the information provided by the mobile device if a server needs to produce more frames in order to catch up or hit the desired frame rate or slow down to do so. After calculating this the controller sends the information again using the publish/subscribe system to the individual servers that calculate the simulation.

The controller can change the speed of calculations at the servers by adjusting the CPU frequencies. If the server needs to speed up then the CPU frequency increases. An increased CPU frequency results then in a faster calculation (or reduced calculation time).

To prove the effectiveness of our approach and evaluate its performance we implemented a proof of concept implementation in Python. This proof of concept implementation uses MQTT for the communication between the servers and the controller.

The evaluation showed that the LQ-CPU-Frequency-Synchronization method is effective to synchronize multiple servers.

Our evaluations also showed that increasing the network delay had in general the effect that the efficiency of the synchronization method is decreased. This means that it takes the LQ-Regulator longer to reach the desired synchronization state. We tested two scenarios where we first used homogeneous hardware and then tested if the results would change if heterogeneous hardware is used. Our results showed that the method worked better when using homogeneous hardware, but

is still able to synchronize heterogeneous hardware. This shows that the synchronization method not only works in servers with different calculation power, but even servers with different CPU architectures.

As part of future work, we propose two aspects that could be subject of further research.

The first direction for future work could be to find correlations between values in the B matrix and CPU capabilities of the servers. Because the LQ-Regulator is not aware of the capabilities of the CPUs it is controlling, it controls all CPUs in the same way. Meaning the controller assumes that by increasing/decreasing the frequency by x, the same speedup or delay is achieved on all (types of) servers. However, increasing the CPU frequency by a value x does not have the same effect on every hardware. The effect is more pronounced on a system with lower hardware specifications, relative to hardware with higher specifications. The B matrix is responsible to control what effect the input has on the system. Because of the different effect on different hardware, the values in B could be chosen differently.

As stated in Section 4.2.1 the currently provided B matrix assumes a change of one MHz in CPU frequency correlates to about a change of one frame per second. This is currently a simplification. If the values in B are decreased, it means that a change in CPU frequency has less impact and therefore the regulator will use higher values to compensate for this. This works for values greater than one in the opposite way, meaning values greater than one will reduce the strength with which the regulator regulates a CPU frequency.

Future work could find a correlation between the CPU capabilities and the values in B to increase the efficiency of the LQ-CPU-Frequency-Synchronization method.

Another direction for future work could be multiple actuating variables. As discussed in Chapter 6 our approach has the limitation that the LQ-Regulator is not able to regulate the servers faster or slower than the minimum or maximum capabilities of the server. Therefore, when deploying the system it has to be considered how complex the calculation is that is run on an individual server. Because the LQ-CPU-Frequency-Synchronization method is continuous, it can handle changes in complexity as shown in Chapter 6. Because of this one could add multiple actuation variables if the CPU frequency is at its limits. For instance, changing to a computation with a different complexity and smaller quality (e.g. less precise result) in addition to the CPU frequency could be investigated.

Bibliography

[AH91] D. Anderson, G. Homsy. “A continuous media I/O server and its synchronization mechanism”. In: *Computer* 24.10 (1991), pp. 51–57. DOI: 10.1109/2.97251 (cit. on pp. 25, 26).

[BGWK13] D. Brodowski, N. Golde, R. J. Wysocki, V. Kumar. “Cpu frequency and voltage scaling code in the linux (tm) kernel”. In: *Linux kernel documentation* (2013), p. 66. URL: https://www.mikrocontroller.net/attachment/529080/Linux-CPU-freq-governors.pdf (cit. on p. 59).

[Boa13] Y. Boas. “Overview of virtual reality technologies”. In: *Interactive Multimedia Conference*. Vol. 2013. 2013 (cit. on p. 15).

[CKY20] E. Chang, H. T. Kim, B. Yoo. “Virtual Reality Sickness: A Review of Causes and Measurements”. In: *International Journal of Human–Computer Interaction* 36.17 (2020), pp. 1658–1682. DOI: 10.1080/10447318.2020.1778351. eprint: https://doi.org/10.1080/10447318.2020.1778351. URL: https://doi.org/10.1080/10447318.2020.1778351 (cit. on p. 32).

[DL97] P. DiZio, J. R. Lackner. “Circumventing side effects of immersive virtual environments”. In: *Advances in human factors/ergonomics* 21 (1997), pp. 893–896 (cit. on p. 32).

[DVFG01] M. H. Draper, E. S. Viirre, T. A. Furness, V. J. Gawron. “Effects of image scale and system time delay on simulator sickness within head-coupled virtual environments”. In: *Human factors* 43.1 (2001), pp. 129–146. DOI: 10.1518/001872001775992552 (cit. on p. 32).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. “The many faces of publish/subscribe”. In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078 (cit. on p. 24).

[EPD94] J. Escobar, C. Partridge, D. Deutsch. “Flow synchronization protocol”. In: *IEEE/ACM Transactions on Networking* 2.2 (1994), pp. 111–121. DOI: 10.1109/90.298430 (cit. on p. 27).

[Fou] E. Foundation. *Eclipse Mosquitto*. URL: https://mosquitto.org/ (cit. on p. 60).

[FZ94] G. Forman, J. Zahorjan. “The challenges of mobile computing”. In: *Computer* 27.4 (1994), pp. 38–47. DOI: 10.1109/2.274999 (cit. on p. 30).

[GG05] J. F. Golding, M. A. Gresty. “Motion sickness”. In: *Current opinion in neurology* 18.1 (2005), pp. 29–34 (cit. on p. 32).

[God03] A. Godhwani. “Feedback control systems”. In: *2003 IEEE Power Engineering Society General Meeting (IEEE Cat. No.03CH37491)*. Vol. 3. 2003, 1758–1761 Vol. 3. DOI: 10.1109/PES.2003.1267423 (cit. on p. 17).

[Han] S. Han. *Debian bullseye cpufrequtils*. URL: https://manpages.debian.org/bullseye/cpufrequtils/index.html (cit. on p. 58).

[HTS08] U. Hunkeler, H. L. Truong, A. Stanford-Clark. "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks". In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE. 2008, pp. 791–798. DOI: 10.1109/COMSWA.2008.4554519 (cit. on p. 24).

[Oas] Oasis. *MQTT 3.1.1 Standard*. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (cit. on pp. 25, 51).

[Olu14] H. S. Oluwatosin. "Client-server model". In: *IOSR Journal of Computer Engineering* 16.1 (2014), pp. 67–71 (cit. on p. 22).

[pyt] python-control-developers@lists.sourceforge.net. *Python control library*. URL: https://pypi.org/project/control/ (cit. on p. 59).

[RVR92] P. Rangan, H. Vin, S. Ramanathan. "Designing an on-demand multimedia service". In: *IEEE Communications Magazine* 30.7 (1992), pp. 56–64. DOI: 10.1109/35.144778 (cit. on p. 27).

[w3r] w3resource. *Python implementation of the Chudnovsky algorihtm*. URL: https://www.w3resource.com/projects/python/python-projects-1.php (cit. on p. 61).

[Wik22] Wikipedia contributors. *Linear–quadratic regulator — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Linear%E2%80%93quadratic_regulator&oldid=1117695476 (cit. on p. 17).

[Wiy] R. R. Wiyatno. *Robotik - Ep.5: Feedback Control with Linear Quadratic Regulator (LQR)*. URL: https://rrwiyatn.github.io/blog/robotik/2020/07/19/lqr.html (cit. on p. 18).

[Wür] J. J. (Würzburg). *Einführung in LQ Regler*. URL: https://www.youtube.com/watch?v=hpzChfVSif4 (cit. on p. 19).

[YLM96] T.-J. Yu, C.-F. Lin, P. Muller. "Design of LQ regulator for linear systems with algebraic-equation constraints". In: *Proceedings of 35th IEEE Conference on Decision and Control*. Vol. 4. 1996, 4146–4151 vol.4. DOI: 10.1109/CDC.1996.577429 (cit. on p. 18).

All links were last followed on January 02, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature