Institute for Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design and Implementation of a Framework to Evaluate Scheduling Algorithms Using Physical Networked Control Systems

David Augustat

Course of Study:	Informatik
Examiner:	Prof. Dr. phil. nat. Christian Becker
Supervisor:	M.Sc. Robin Laidig, Dr. rer. nat. Frank Dürr
Commenced:	September 22, 2022
Completed:	March 22, 2023

Abstract

Networked Control Systems (NCS) are commonly used in industrial applications like telerobotics, smart energy grids, and autonomous vehicles. In many cases, NCS share their network with other participants competing for the available bandwidth. This necessitates scheduling algorithms respecting the time-critical nature of control systems. Scientific evaluations under reproducible conditions are required to assess the performance of a given scheduling algorithm in the context of networked control systems. In this thesis, a framework to evaluate the performance of scheduling algorithms using a physical networked control system is designed and implemented. The framework comprises an inverted pendulum connected to an IEEE 802.3 Ethernet network featuring a software switch. The software switch can be programmed to execute arbitrary scheduling algorithms, significantly simplifying the evaluation process. This thesis explains the framework's design, implementation, and usage in detail. We use the proposed framework to evaluate the Multi-priority Token Bucket scheduling approach (MPTB) designed at the Institute for Parallel and Distributed Systems (IPVS) of University of Stuttgart. This scheduling algorithm dynamically assigns priorities to data streams according to their contract compliance. It is found that MPTB can provide better stability to the inverted pendulum at a lower average data rate than traditional FIFO scheduling. However, we also find that the selection of parameters for MPTB severely impacts the scheduling algorithm's performance. Further, we find that using real cross-traffic to stress the network yields non-deterministic latencies, while simulated delays at the software switch are better suited for reproducible evaluations.

Kurzfassung

Vernetzte Regelungssysteme (Networked Control Systems, NCS) werden häufig in industriellen Anwendungen wie der Telerobotik, intelligenten Stromnetzen und autonomen Fahrzeugen eingesetzt. In vielen Fällen teilen sich die NCS ihr Netz mit anderen Teilnehmern, die um die verfügbare Bandbreite konkurrieren. Dies erfordert Schedulingverfahren, die den zeitkritischen Charakter von Regelungssystemen berücksichtigen. Wissenschaftliche Messungen unter reproduzierbaren Bedingungen sind erforderlich, um die Leistung eines bestimmten Schedulingverfahrens im Kontext vernetzter Regelungssysteme zu beurteilen. In dieser Arbeit wird ein Rahmenwerk zur Bewertung der Leistung von Scheduling-Algorithmen unter Verwendung eines physischen vernetzten Regelungssystems entworfen und implementiert. Das Rahmenwerk besteht aus einem inversen Pendel, das an ein IEEE 802.3 Ethernet-Netzwerk mit einem Software-Switch angeschlossen ist. Der Software-Switch kann so programmiert werden, dass er beliebige Scheduling-Algorithmen ausführt, was den Evaluierungsprozess erheblich vereinfacht. In dieser Arbeit wird der Entwurf, die Implementierung und die Verwendung des Rahmenwerks im Detail erläutert. Wir verwenden das vorgeschlagene Rahmenwerk, um den Multi-Priority-Token-Bucket-Schedulingansatz (MPTB) zu evaluieren, welcher beim Institut für Parallele und Verteilte Systeme (IPVS) an der Universität Stuttgart entworfen wurde. Dieses Schedulingverfahren weist den Datenströmen dynamisch Prioritäten entsprechend ihres Verhaltens zu. Es zeigt sich, dass MPTB bei einer niedrigeren durchschnittlichen Datenrate eine bessere Stabilität des Pendels erzielen kann als das klassische FIFO-Scheduling. Wir stellen jedoch auch fest, dass die Auswahl der Parameter für MPTB großen Einfluss auf die Performanz des Scheduling-Algorithmus' hat. Außerdem zeigt sich, dass die

Verwendung von echtem Cross-Traffic zur Auslastung des Netzwerks zu instabilen Latenzen führt, während simulierte Verzögerungen am Software-Switch besser für reproduzierbare Messungen geeignet sind.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

ATS Asynchronous Traffic Shaping.

CAN Controller Area Network.

CPU Central Processing Unit.

CSR Constant Sampling Rate.

DEI Drop Eligibility Indicator.

DSR Dynamic Sampling Rate.

FIFO First in, first out.

GPIO General Purpose Input/Output.

GPU Graphics Processing Unit.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

IO Input/Output.

IP Internet Protocol.

JSON JavaScript Object Notation.

LAN Local Area Network.

LQR Linear Quadratic Regulator.

MPTB Multi-Priority Token Bucket.

NCS Networked Control System.

OS Operating System.

PCI Peripheral Component Interconnect.

PCP Priority Code Point.

RAM Random-Access Memory.

SATA Serial AT Attachment.

SDN Software-Defined Networking.

SKB Socket Buffer.

SP Strict Priority.

SPI Serial Peripheral Interface.

SSH Secure Shell.

TCP Transmission Control Protocol.

UART Universal Asynchronous Receiver / Transmitter.

UDP User Datagram Protocol.

USB Universal Serial Bus.

VLAN Virtual Local Area Network.

1 Introduction

Networked control systems (NCS) are commonly used in industrial applications, for example, telerobotics [KAF+15], smart energy grids, and autonomous vehicles [Tec23]. NCS are control systems regulated through a control loop that is closed over a network. Networks used for NCS often face multiple data streams competing for the available bandwidth. This necessitates network scheduling at the switches of the network. As networked control systems are time-critical applications, it is essential to use scheduling algorithms that guarantee low latencies and predictable latency bounds. For this purpose, scheduling algorithms must be evaluated for their suitability for NCS.

A popular example of a networked control system is a network-controlled inverted pendulum [SWVP17] [Nie19] [Car22]. This system consists of an upwards-pointing rod on a movable cart. A controller moves the cart on a track to prevent the rod from tipping over due to gravitational force. A network is used to transmit the state of the inverted pendulum to the controller. As the inverted pendulum is a highly unstable non-linear system, the transmission via the network is time-critical. A high end-to-end latency can lead to instability of the system. Therefore, scheduling algorithms used for switches within the network must ensure low latency.

There already exists a lot of research on the use of scheduling algorithms with network-controlled inverted pendulums [SWVP17] [Sch16] [Nie19]. Many works do not consider real networks and real inverted pendulums but instead rely on simulations [Zin16] [Nie19]. While this is often easier and less prone to measurement errors due to determinism, such simulations may not consider the complexity of the real world. Thus, simulations can have idealistic assumptions that are not present in real-world applications. There are also studies covering the evaluation of scheduling algorithms with physical inverted pendulums [SWVP17] [Car22]. However, most researchers design their evaluation system for only one or a few fixed scheduling algorithms. This implies that evaluating arbitrary scheduling algorithms with these existing systems can be difficult.

Multi-priority Token Bucket (MPTB) is a dynamic priority-based scheduling algorithm designed for data streams with varying bandwidth consumption. It was created at the Institute for Parallel and Distributed Systems (IPVS) of University of Stuttgart. The algorithm provides multiple priority classes for network traffic, each with a different per-hop latency guarantee. Each data stream gets dynamically assigned one of the provided priority classes. For this purpose, each stream specifies a contract consisting of a desired priority class and a token bucket, which determines the stream's sustainable data rate and maximum burst size. As long as the stream complies with its contract, it gets assigned its desired priority class. However, the stream is also allowed to violate its contract. In this case, the stream's priority gets temporarily reduced according to the severity of the contract violation. MPTB differs from traditional scheduling approaches with guaranteed latency bounds, as the sender may violate its contract temporarily.

This thesis aims to design and implement a framework to evaluate arbitrary scheduling algorithms using a physical networked control system. For this purpose, an existing physical inverted pendulum is integrated into a physical IEEE 802.3 Ethernet network, including a programmable software switch. The software switch can be reprogrammed conveniently to run arbitrary scheduling algorithms. The framework also involves the generation of cross traffic, which stresses the switch to evaluate the performance of scheduling algorithms under load. Alternatively, a load on the switch can be simulated using artificial delays. The framework also provides a logging infrastructure to record the behavior of the network and the inverted pendulum during evaluations. The constructed framework can be used in future research to evaluate the performance of scheduling algorithms in the scope of networked control systems without having to create such a system from the ground up. Therefore, this framework can make future research in this area significantly easier and faster. Another goal of this work is to assess the performance of the MPTB scheduling algorithm for networked control systems. For this purpose, evaluations with MPTB and other scheduling algorithms are conducted with the constructed framework.

The rest of this thesis is structured as follows: In Chapter 2, several fundamental concepts and technologies used in this work are introduced. Afterward, Chapter 3 encompasses existing research related to this paper's scope. In Chapter 4, the system model assumed in this work, and the problem statement are given. Then, the design of the framework and its components, as well as the Multi-priority Token Bucket scheduling algorithm are discussed in Chapter 5. The implementation of this design is then explained in Chapter 6. Next, the performance of MPTB is compared to that of FIFO scheduling within multiple evaluations in Chapter 7. Finally, Chapter 8 summarizes the results of this work and provides an outlook.

2 Background

There are several concepts and technologies that are important for this work. This section briefly introduces these. First, the concepts of networked control systems and inverted pendulums are explained. Then, the involved microcontrollers and single-board computers are introduced. Afterward, multiple network-related concepts and networking in Linux are discussed.

2.1 Networked Control System

In a (traditional) control system, the system's behavior is regulated through a control loop. That is, the system produces an output state $y(t)$, which a controller uses to calculate an input $u(t)$, which is then applied to the system to alter its state [HL05].

A *networked* control system (NCS) is a control system whose feedback loop is closed by a network. The network is usually considered a communication channel shared with other nodes not involved in the control system [GC08]. For instance, such a network can be the internet or a local Ethernet network.

Contrary to a traditional control system, the communication channel in an NCS does not necessarily have real-time guarantees and may introduce non-deterministic behavior and latency [Sie17]. This places an additional difficulty, as network influences can have an impact on the performance of the NCS.

From a structural view, an NCS consists of a plant, a controller, and a communication medium. The plant is a to-be-regulated device with an output $y(t)$ that depends on the current time t. This output is sent to the controller via the communication medium. However, the controller receives an altered state $\overline{y}(t)$ since the network may modify the state, e.g., by introducing latency. The controller then calculates an input (which we refer to as "control instruction") $\overline{u}(t)$ and sends this value to the plant via the network. The plant then receives an altered input $u(t)$, which it uses to adapt its state [HL05]. A graphic displaying this procedure can be seen in Figure 2.1. This structure has variants where only the plant output or the control input is sent via the network.

2.2 Inverted Pendulum

A common example of a networked control system is an inverted pendulum. An inverted pendulum is a special kind of pendulum that has its center of mass *above* its pivot point. Commonly, the pivot point of the pendulum is formed by a cart that can move left and right on a track. A pole is mounted to the cart, pointing upwards. This system is unstable as the pole will fall over due

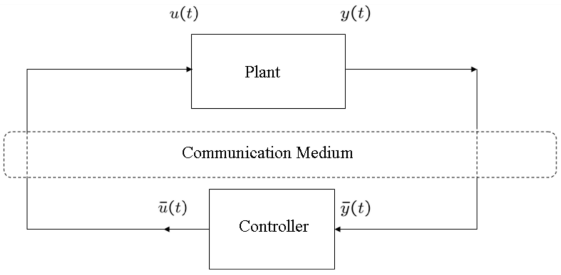


Figure 2.1: Model of a networked control system. $y(t)$ is the output of the plant and $u(t)$ the input. $\overline{y}(t), \overline{u}(t)$ can differ from $y(t), u(t)$ as they may have been altered by the network, e.g., by the introduction of a delay. Source: [HL05]

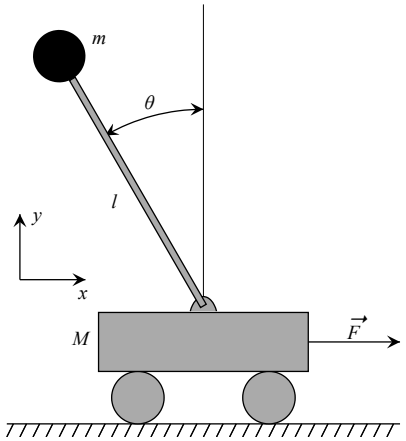


Figure 2.2: Inverted pendulum on a cart. Source: [Wik12]

to gravity. Therefore, a controller permanently needs to monitor the angle of the pole and move the cart accordingly to prevent the pole from falling over. The structure of a cart-based inverted pendulum is shown in Figure 2.2.

The state of an inverted pendulum can be described by the vector

$$(x, \dot{x}, \theta, \dot{\theta})^T$$

where x and $\dot{x}$ represent the position and velocity of the cart and θ and $\dot{\theta}$ represent the angle and angular velocity of the pole [Nie19]. In this work, we always assume that the center of the track has position $x = 0$ and the upwards position of the pole has angle $\theta = 0°$. Thus, the controller's goal is to achieve the optimal state $(0, 0, 0, 0)^T$ where the pole is upwards, the cart is in the center, and both the pole and cart are not moving.

2.3 Teensy 3.6 Development Board

The Teensy 3.6 development board [PJR23a] is a microcontroller based on the MK66FX1M0VMD18 ARM Cortex-M4 processor. It runs at 180 MHz and features 1024 kB of flash memory plus 1024 kB of RAM. The board has 64 digital I/O pins, of which 22 also support pulse-width modulation,

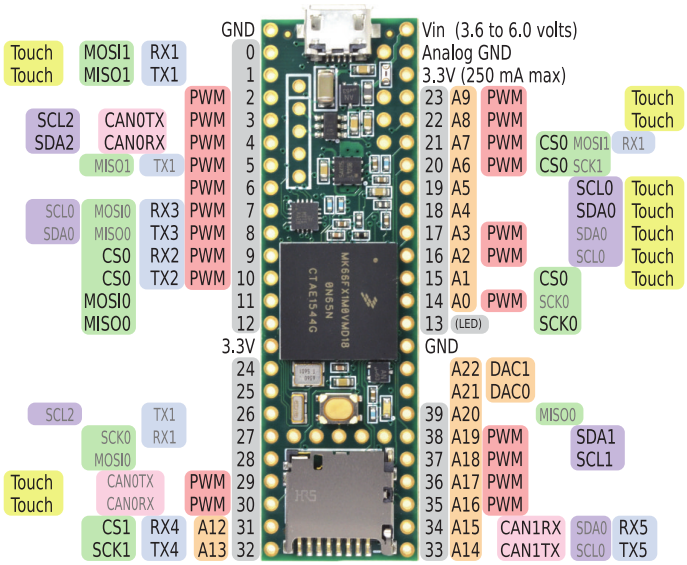


Figure 2.3: Pinout of the Teensy 3.6 microcontroller board. Source: [PJR23a]

25 can take analog input, and two pins support analog output. The board has a USB interface and supports SPI, CAN, serial, and I2C for communication with peripherals. A picture of the Teensy 3.6, including its pinout, can be seen in Figure 2.3.

The microcontroller supports Teensyduino [PJR23b], an add-on for the Arduino platform that allows running Arduino sketches on Teensy boards. Arduino is a development platform for microcontrollers that can be used to write programs (so-called "sketches") in a variant of C++. These sketches can then be compiled and uploaded onto the microcontroller through a simple graphical interface [Ard18].

2.4 Raspberry Pi 2 / CM4, ODROID C2, Banana Pi M2 Berry

The Raspberry Pi 2, Raspberry Pi CM4, ODROID C2, and Banana Pi M2 Berry are all single-board computers based on the ARM processor architecture. All four devices can run lightweight distributions of Linux. In the following, the devices will be introduced briefly.

2.4.1 Raspberry Pi 2 (Model B)

The 2015-released Raspberry Pi 2 Model B has a 900MHz quad-core ARM Cortex-A7 CPU and 1 GB of RAM. It features a 100 Mbit/s Ethernet interface, which is lower than the other single-board computer that all feature a 1 Gbit/s interface [Ras23b]. As an operating system, we installed Ubuntu Server 22.04.

2.4.2 Raspberry Pi CM4

The Raspberry Pi Compute Module 4 is a compact single-board computer that must be mounted on a Compute Module 4 IO Board for operation. The computer features a Broadcom BCM2711 quad-core Cortex-A72 ARMv8 64-bit processor and 2 GB of RAM. The device has a 16 GB flash memory which means that an external SD card is not required as a boot medium [Ras23c]. The IO board provides a 1 Gbit/s Ethernet port. As an alternative to that, the IO Board also has a PCI Express connector which can be used to attach a multi-port Ethernet card. As an operating system, the CM4 considered in this work runs Raspbian 11.

2.4.3 ODROID C2

The ODROID C2 features an Amlogic S905 Quad Core Cortex-A53 ARMv8 64-bit processor, 2 GB of RAM, and a Mali-450 GPU. A 1 Gbit/s Ethernet port is present. As the name suggests, the ODROID C2 can run Android; however, no official images with a higher version than Android 6.0 are available [ODR23]. Instead, for this work, we installed the Armbian 22, which is a command-line-only port of Ubuntu 22.04 [Arm23].

2.4.4 Banana Pi M2 Berry

The Banana Pi M2 Berry features an Allwinner A40i Quad Core ARM Cortex A7 CPU and 1GB of RAM. It has a 1 Gbit/s Ethernet interface and a SATA port [Ban22]. As an operating system, we installed Armbian 21.02, a command-line-only port of Ubuntu 20.04.

2.5 Token Bucket Traffic Shaper

A token bucket is a traffic shaper where each transmitted byte is associated with a token. Bytes can only be sent as long as there are tokens left in a bucket.

More formally, the traffic shaper consists of a bucket that can hold b tokens. The bucket is constantly filled with tokens at rate r [tokens/second], but all tokens that would overflow the bucket's capacity b are discarded. When the sender sends a byte, a token is extracted from the bucket if the bucket is not empty. If the sender wants to send a byte while the bucket is empty, this is considered a violation of the contract. In this case, the packet containing this byte is either discarded, flagged as non-conforming, or held back until enough tokens are in the bucket.

This construct limits the maximum burst size to b, meaning a sender can send at most b bytes at once. On average, over time, the sender can only transmit at the rate r [Tan11] [Rot21]. This concept is illustrated in Figure 2.4.

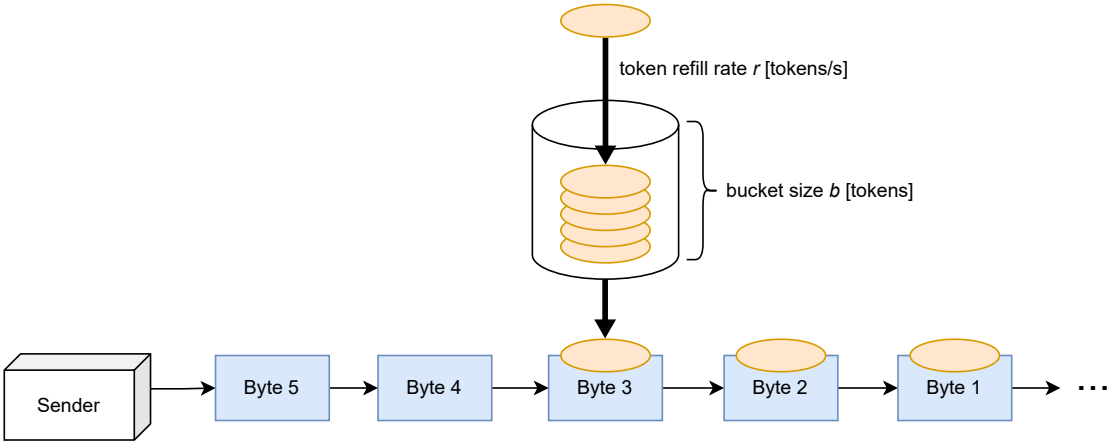


Figure 2.4: Illustration of a token bucket traffic shaper. Only bytes that were assigned a token may be sent as conforming traffic.

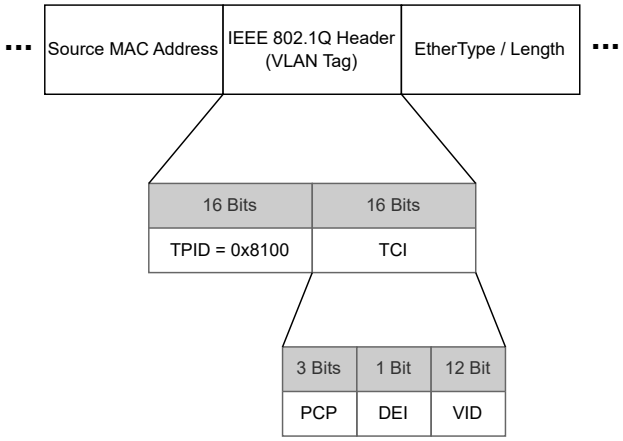


Figure 2.5: Structure of the IEEE 802.1Q Header. Graphic inspired by [Her21].

2.6 Ethernet VLAN Tagging

IEEE 802.1Q [IEE22] specifies virtual local area networks (VLANs), which can be used to divide a single physical local area network (LAN) into multiple discrete virtual LANs. This works by attaching a specific VLAN tag, formally called the IEEE 802.1Q Header, to every frame transmitted inside a VLAN. This tag, among other things, contains a 12-bit VLAN identifier (VID) that marks to which VLAN the frame belongs. When a switch inside the physical LAN sees a frame with a VLAN tag, it handles the frame according to the information from the tag.

Most important for the scope of this work, the VLAN tag includes a 3-bit priority code point (PCP) field. Senders can use this field to set a priority value. The switch can then handle the frame according to its PCP value. How exactly the switch handles frames depending on their PCP is not part of the specification but depends on the implementation of the switch. Figure 2.5 shows the structure of the VLAN tag.

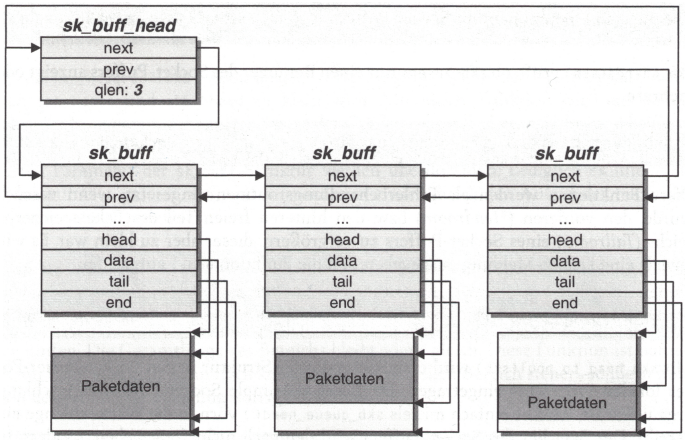


Figure 2.6: Multiple socket buffers connected to each other through pointers to form a queue. Source: [Weh02]

2.7 Network Scheduling in Linux

In Linux, the handling and scheduling of network packets are part of the kernel. Packets are stored using the *socket buffer* data structure, and a *queueing discipline* conducts the scheduling. In this section, these concepts will be introduced. It is based on information from Wehrle et al. [Weh02] and the official Linux 5.15 kernel source [Lin23].

2.7.1 Socket Buffer Data Structure

The socket buffer (`sk_buff`) is a general data structure in Linux for network packets of any sort. It stores the packet's payload (using pointers to memory) and various metadata fields. Socket buffers also have `next` and `previous` fields, which can be used to create a double-linked list of multiple socket buffers. Scheduling algorithms use these fields to form queues. A visualization of this concept can be seen in Figure 2.6.

For the scope of this work, the `priority` field (in this work referred to as `skb->priority`) is the most important one. It stores an unsigned 32-bit integer indicating the priority of the packet. This priority field is accessible from within Qdiscs (see Section 2.7.2) and can thus be used to handle packets differently based on their priority.

2.7.2 Qdiscs (Queueing Disciplines)

In Linux, so-called Qdiscs (queueing disciplines) are used to implement network scheduling algorithms. Qdiscs can, however, also be used for traffic shaping and policing [Hub22]. Qdiscs are usually written in C and need to offer the functions specified in the `Qdisc_ops` struct shown in Listing 2.1. Most importantly, a Qdisc offers the following functions:

Listing 2.1 `Qdisc_ops` data structure in the Linux Kernel specifying the structure of a Qdisc [Lin23]

```
struct Qdisc_ops {
  struct Qdisc_ops *next;
  const struct Qdisc_class_ops *cl_ops;
  char id[IFNAMSIZ];
  int priv_size;
  unsigned int static_flags;

  int (*enqueue)(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **to_free);
  struct sk_buff * (*dequeue)(struct Qdisc *);
  struct sk_buff * (*peek)(struct Qdisc *);

  int (*init)(struct Qdisc *sch, struct nlattr *arg, struct netlink_ext_ack *extack);
  void (*reset)(struct Qdisc *);
  void (*destroy)(struct Qdisc *);
  int (*change)(struct Qdisc *sch, struct nlattr *arg, struct netlink_ext_ack *extack);
  void (*attach)(struct Qdisc *sch);
  int (*change_tx_queue_len)(struct Qdisc *, unsigned int);
  void (*change_real_num_tx)(struct Qdisc *sch, unsigned int new_real_tx);

  int (*dump)(struct Qdisc *, struct sk_buff *);
  int (*dump_stats)(struct Qdisc *, struct gnet_dump *);

  void (*ingress_block_set)(struct Qdisc *sch, u32 block_index);
  void (*egress_block_set)(struct Qdisc *sch, u32 block_index);
  u32 (*ingress_block_get)(struct Qdisc *sch);
  u32 (*egress_block_get)(struct Qdisc *sch);

  struct module *owner;
};
```

- `int enqueue(struct sk_buff *skb, ...)`: Passes the packet stored in `skb` to the scheduling algorithm. Many scheduling algorithms use queues to store packets. Hence it is called "enqueue". An error code is returned if the Qdisc cannot accept the packet, e.g., due to congestion.
- `struct sk_buff* dequeue(...)`: The Qdisc returns a pointer to the packet that should be sent next. If no packet is available for transmission, `NULL` is returned instead.
- `int init(...)`: Initialized the Qdisc.
- `void reset(...)`: Resets the Qdisc to its initial state. The Qdisc also calls `reset(...)` on all of its child Qdiscs in case they exist.
- `void destroy(...)`: Releases all resources that were acquired by the Qdisc.

Combining multiple Qdiscs in a tree-like structure is possible: One Qdisc can have multiple child Qdiscs, but every Qdisc has exactly one parent. There is always one Qdisc marked as `root`, which is the root node of this tree. Whenever the kernel wants to transmit a packet, its SKB pointer is passed to the `root` Qdisc. Each Qdisc can then pass the pointer to one of their child Qdiscs [Hub22].

Listing 2.2 Example usage of `tc` to set up Qdiscs for the `ens3` interface: Priority scheduling with three queues, where each queue is associated with a different token bucket for traffic shaping.

```
sudo tc qdisc add dev ens3 root handle 42 prio bands 3
sudo tc qdisc add dev ens3 parent 42:1 handle 421 tbf rate 1mbit burst 1mbit limit 1mbit
sudo tc qdisc add dev ens3 parent 42:2 handle 422 tbf rate 2mbit burst 2mbit limit 1mbit
sudo tc qdisc add dev ens3 parent 42:3 handle 423 tbf rate 3mbit burst 3mbit limit 1mbit
```

For instance, one can use a priority scheduling (`prio`) with three priorities as the root Qdisc and then attach three different token bucket Qdiscs (`tbf`) to the individual queues of the `prio` Qdisc. This way, different priorities can be given different bandwidth contracts.

The configuration of Qdiscs from the command line is carried out via the `tc` (traffic control) command line tool. For instance, the example from the previous paragraph can be realized with the Shell commands from Listing 2.2. There it can be seen that every Qdisc instance gets assigned a handle (32-bit integer) which can then be used to refer to the Qdisc when attaching child Qdiscs or to modify it. The sub-queues of a Qdisc with n queues are numbered from 1 to n and can be accessed with the handle `handle:queueNumber`.

2.8 Linux Network Configuration With `ip`

In Linux, network interfaces can be configured with the `ip` command line tool. This tool can create VLAN interfaces and virtual bridges, change interface settings, assign IP addresses, and turn interfaces on or off, among many other functions [Lit22]. The tool can also be used to display current network configurations, which can be helpful for troubleshooting.

An example showing the usage of `ip` commands to set up a software switch on Linux can be seen in Listing 6.5.

3 Related Work

In this chapter, various related studies will be introduced briefly. First, the strict priority scheduling algorithm from Grigorjew [GMH+20] is presented. Afterward, an inverted pendulum study by Schindler et al. [SWVP17] is discussed. Followed by this, multiple studies involving inverted pendulums that were conducted at the University of Stuttgart will be introduced. Finally, the differences between existing research and this thesis are given.

3.1 Grigorjew et al.: Strict Priority Scheduling

Grigorjew et al. [GMH+20] present a strict-priority multi-queue scheduling algorithm with provable per-hop latency bounds. The algorithm only requires local information at each switch and does not need a central network controller.

The authors consider a network of switches through which Ethernet frames can be sent. Each frame belongs to a stream associated with a discrete traffic class (priority).

When a sender wants to establish a new stream, resources at the switches participating in this stream must be reserved. For this purpose, the sender transmits the following information: Traffic class (static), minimum and maximum frame length, and minimum and maximum accumulated latency. Additionally, the sender specifies its traffic contract by communicating its burst size b_i (maximum number of bits the sender may send at once) and the minimal time interval τ_i between two bursts. Each switch along the route of the stream then checks whether it can guarantee the latency requirements for the specified stream in the requested traffic class (for details, see [GMH+20]). The switches can make their decision independently without talking to a central controller. If every switch accepts the stream, the stream gets admitted to the network, and the resources are reserved.

At each switch, there is a FIFO queue for each traffic class. When a frame arrives at a switch, it gets inserted into the queue of its corresponding traffic class. When selecting a frame for transmission, the switch always sends frames from higher classes first. A frame of class i will not be sent as long as the queues for classes $> i$ still contain untransmitted frames.

The authors show that this scheduling algorithm has a deterministic maximum per-hop latency for each traffic class as long as the following constraints hold:

- The sender never exceeds its communicated burst size b_i and maximum frame size $\hat{\ell}_i$
- The sender never exceeds their long-term data rate $r_i := b_i/\tau_i$.

This means the senders are not allowed to violate their traffic contract specified by the burst rate b_i and the minimal time interval τ_i between bursts.

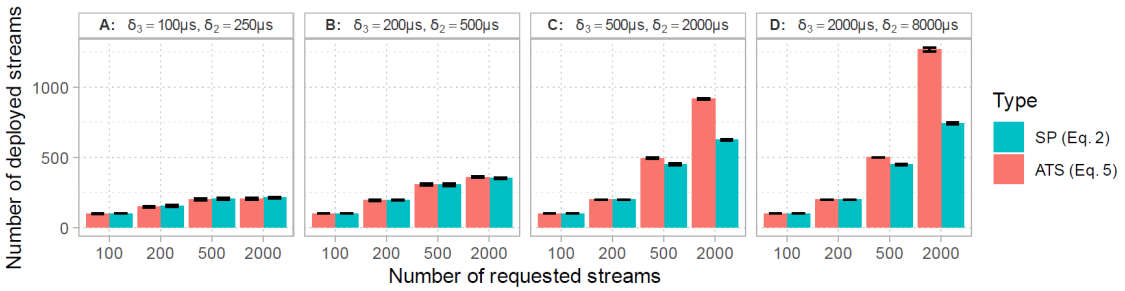


Figure 3.1: Results of the comparison between SP and ATS in Grigorjew et al. Source: [GMH+20]

The authors compare their scheduling algorithm, which they call SP for "strict priority", to another algorithm called Asynchronous Traffic Shaping (ATS). ATS is a similar approach, where a token bucket traffic shaper processes each stream at each switch. After the traffic shaping has been applied, the packets of each stream get inserted into one of multiple FIFO queues according to their traffic class [IEE20]. ATS is of interest because it also has deterministic per-hop latency bounds but requires special hardware in the switch for traffic shaping. In contrast, SP does not require special hardware but works with existing switches.

The authors evaluate SP and ATS in a simulated network with multiple senders and receivers connected through switches. Different per-hop delay guarantees are used for these switches. The researchers use the ratio of accepted streams to requested streams as a performance metric. Thus, in the optimal case, the number of accepted streams equals the number of requested ones.

It was found that ATS always performs at least as well as SP. ATS and SP performed very similarly for tight delay guarantees, whereas for looser delay guarantees, ATS performed up to 70% better than SP. Figure 3.1 shows the results in more detail. The experiment has also shown that traffic shaping with ATS does not yield any benefit if the end-to-end latency of a stream is smaller than its maximum burst interval, as bursts cannot accumulate in this situation.

3.2 Schindler et al.: Wireless Network for Inverted Pendulum

Schindler et al. [SWVP17] implemented a 6TiSCH multi-hop network for the wireless transmission of sensor data from an inverted pendulum. 6TiSCH (IPv6 over Time Slotted Channel Hopping) [Thu21] is a network architecture for low-power wireless devices providing low latency, low jitter, and high reliability. In 6TiSCH, the time is divided into time slots of a fixed length. A periodic schedule specifies for each device whether it should transmit, listen or sleep in a given time slot. Besides the time slots, multiple frequency bands ("channels") exist, which the devices can transmit on.

The researchers use a simple scheduling strategy for time slot selection where each node uses exactly the same schedule. In an active time slot, a node transmits if it has data to send; otherwise, it listens for incoming packets. In non-active slots, all nodes switch to sleep mode. As all nodes are active at the same time, collisions are possible. These are resolved by a back-off mechanism not further specified. For the selection of the channel, a pseudo-random generator function is used.

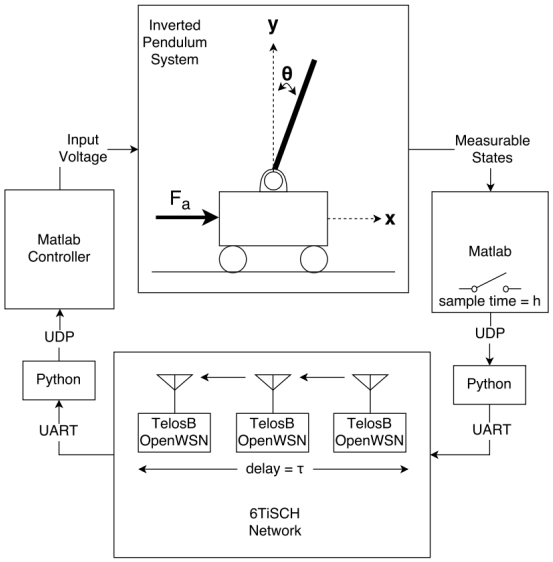


Figure 3.2: Setup of the experiment conducted by Schindler et al. Source: [SWVP17]

The authors first measured the impact of hops, active slots, and radio interference on the end-to-end latency of the network. For this purpose, a network of multiple TelosB microcontrollers was used without the inverted pendulum. They found that a larger number of hops and a lower number of active time slots correlate with higher end-to-end latency. There is no clear correlation between radio interference and latency. For all tested scenarios, the median end-to-end latencies range from 47 to 130 milliseconds, suggesting that 6TiSCH is suitable for usage with an inverted pendulum.

Next, Schindler et al. created a test system where the sensor of a physical inverted pendulum transmits its measurement values to the actor through a 6TiSCH network consisting of multiple TelosB microcontrollers. Figure 3.2 shows the system's detailed structure. The experiment was conducted for a network with one and two hops to measure the significance of hop count. For comparison, the network was replaced by a wired connection in one test run. The pendulum's minimum and maximum angle was used as a performance metric. It was found that the wired connection performed best (-0.88° to 0.88°), the network with one hop performed slightly worse (-1.67° to 2.02°), and the two-hop network performed worst (-2.46° to 2.11°). However, the pendulum was stable in all tested configurations, yielding that a wireless 6TiSCH network is suitable for controlling an inverted pendulum.

3.3 Inverted Pendulum Studies at University of Stuttgart

The inverted pendulum has extensively been studied as a networked control system at the University of Stuttgart. In the following, multiple theses that either form the basis for or are related to this thesis are introduced.

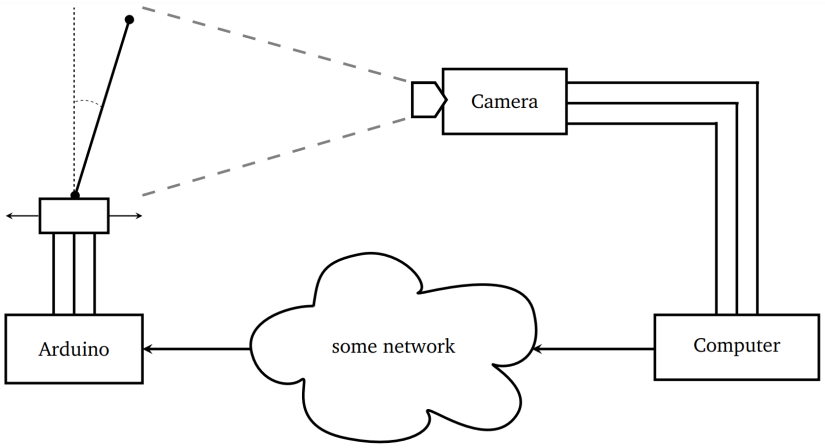


Figure 3.3: Setup of Nägele's inverted pendulum system. Source: [Näg15]

3.3.1 Daniel Nägele's Bachelor's Thesis

Nägele built an inverted pendulum with an optical feedback system [Näg15]. He re-purposed the carriage of a desktop printer as the base of an inverted pendulum and used an Arduino Due microcontroller as a controller for the carriage's stepper motor. Nägele used a PlayStation Eye camera to sense the angle of the pendulum. The camera is connected to a computer that calculates the pendulum's angle from the camera's images using computer vision. The computer is connected to the Arduino Due via an arbitrary network to transmit the measured angles. A diagram of the setup can be seen in Figure 3.3.

For the network between the computer and the Arduino, a serial connection via USB, SPI, CAN, or Ethernet is considered. The serial connection had round-trip times of ≤ 0.61 ms in 99% of the transmissions. SPI and CAN were not tested due to incompatibilities with the computer. For Ethernet, round-trip times of ≤ 1.27 ms could be achieved in 91.4% of transmissions. However, the author states that the comparably poor performance of Ethernet is mainly caused by the aged WIZNet W5100 Ethernet Controller board that was used in the experiment. Nägele assumes that the performance of Ethernet could be improved by using the integrated Ethernet hardware module of the Arduino Due's processor.

3.3.2 Benjamin Schirle's Bachelor's Thesis

While the inverted pendulum created by Nägele [Näg15] used a camera as its sensor, Schirle's work [Sch16] aims at constructing an inverted pendulum that uses an accelerometer and gyroscope to determine the angle of the pendulum.

Schirle uses a Bosch XDK110 development board, which features an accelerometer and a gyroscope and is equipped with a WiFi module [Bos17]. In the author's construction, this XDK110 is attached to the top of the inverted pendulum. This board measures the acceleration and orientation sensor values, encodes them as a UDP packet, and sends this packet to a Raspberry Pi via a WiFi access point. The Raspberry Pi is connected to the stepper motor of the inverted pendulum and uses the received values to control the motor. A diagram of this system model is shown in Figure 3.4.

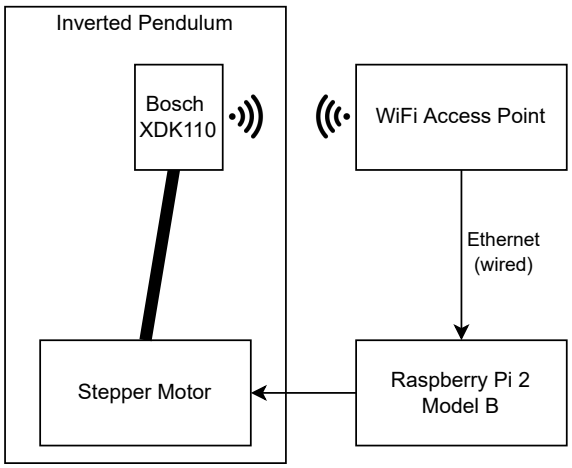


Figure 3.4: Structure of Schirle's inverted pendulum system

As the raw accelerometer and gyroscope sensor values from the XDK110 suffer from noise, the author apply either a complementary filter or a Kalman filter to the values. The filtered values are then used to calculate the movements of the stepper motor. Results from the experiment revealed that the Kalman filter's output was too noisy and thus unacceptable. The complementary filter performed better, but its output still contained too much noise. It was also found that the complementary filter's output reacted faster to changes in the input than the Kalman filter.

It was not achieved to stabilize the inverted pendulum with the tested filters. Instead, the author suggests that the average of two updates might be used as an input for the controller unit, even though this was not tested. It is also mentioned that a different motor might help stabilize the pendulum.

Besides this, the author found that the system is capable of transmitting one sensor update (as a UDP packet) to the receiver every ten milliseconds.

3.3.3 Stephan Zinkler's Master's Thesis

Zinkler's master's thesis [Zin16] is about packet scheduling algorithms for use with multiple networked control systems in an IP network. The network consists of multiple NCS, network switches, and unrelated non-time-sensitive applications. The author points out that a naive scheduling approach separating traffic into time-sensitive and non-time-sensitive packets is insufficient for this use case, as there are multiple NCSs. Instead, Zinkler suggests a software-defined networking (SDN) approach with priority scheduling.

From a high-level view, the priority-based scheduling mechanism works as follows: All time-sensitive packets from NCSs are marked as such using a flag in their header. Time-sensitive packets also include a priority value which is determined by the sensor component of the NCS. Whenever a packet arrives at a switch, this header flag is checked. If the time-sensitive flag is set, the packet is put into a priority queue according to its included priority value. Otherwise, the packet gets inserted into a FIFO queue with only best-effort guarantees. Every time the switch is ready to send a packet,

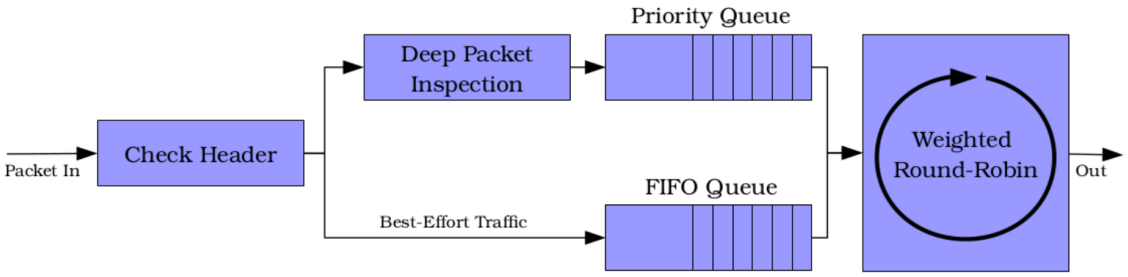


Figure 3.5: Packet flow with Zinkler's scheduling approach. Source: [Zin16]

the packet gets extracted from either the priority queue (time-sensitive traffic) or the FIFO queue (best-effort traffic) according to a weighted round-robin scheme. This packet flow is depicted in Figure 3.5.

However, the priorities of the time-sensitive packets are not necessarily static. Instead, they can dynamically be modified at each switch using OpenFlow [Ope13], a software-defined networking protocol for dynamic packet routing. With OpenFlow, all switches are connected to an SDN controller, which provides sets of rules (so-called "flow tables") to the switches. The flow tables then instruct the switches about how to modify the priorities of the packets.

For the evaluation, Zinkler simulated multiple inverted pendulums (NCS) and a virtual network consisting of switches. Despite virtual OpenFlow switches being used, the flow tables did not modify the priorities of the packets dynamically. Instead, a dynamic modification of the priorities depending on the current network state is suggested as future work.

The author compared the previously introduced priority scheduling algorithm to a naive round-robin scheduling approach in a network involving six inverted pendulums. He found that the priority scheduling approach could stabilize the inverted pendulums better and faster than the round-robin approach.

3.3.4 Adriaan Nieß's Bachelor's Thesis

Nieß compares the performance of several network scheduling algorithms in the context of an inverted pendulum [Nie19]. For this purpose, Nieß simulates a network where an inverted pendulum's sensor and controller are connected through an IEEE 802.1Qbv [IEE16] compliant Ethernet switch. Four other devices generating traffic are also connected to the switch. The network structure is shown in Figure 3.6. For network simulation, the NeSTiNg framework [FHC+19], which is based on the OMNeT++ network simulation library is used. The pendulum controller calculates its output values from the sensor values using a linear-quadratic regulator (LQR).

Nieß considers two different policies applied in case of packet loss: The hold policy and the zero policy. With the hold policy, the pendulum controller re-uses the last received sensor value if no value arrives in a sampling period. With the zero policy, the value "0" is used instead.

The author evaluated three scheduling algorithms whereby each algorithm was tested with zero policy and hold policy:

- No prioritization of time-sensitive packets

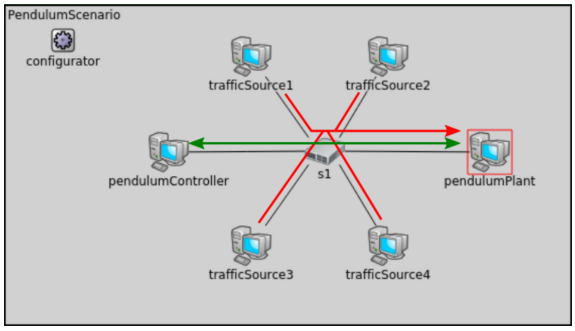


Figure 3.6: Network structure in Nieß's evaluation. Source: [Nie19]

- Priority scheduling
- Priority scheduling with IEEE 802.1Qbv time-aware shaper (decreases interference of time-sensitive traffic with cross-traffic)

Each of the six configurations was evaluated with no, low, medium, high, and very high cross-traffic from the traffic source devices. The cost function of the linear-quadratic regulator ("LQR cost") controlling the pendulum was used as a metric for the performance of the configurations.

It was found that the pendulum becomes unstable at high cross-traffic when no prioritization of time-sensitive traffic is used. For priority scheduling without a time-aware shaper, the pendulum was stable even at very high network congestion when used with the hold policy. However, with the zero policy, the system was only stable at medium congestion. When priority scheduling with an IEEE 802.1Qbv time-aware shaper was used, the pendulum was stable at high congestion with both the hold and zero policies.

Concerning the LQR cost metric, the same result is obtained: The scheduling algorithm without prioritization performed the worst, priority scheduling without a time-aware shaper performed better, and priority scheduling with an IEEE 802.1Qbv time-aware shaper performed the best.

3.3.5 Ben Carabelli's Doctoral Thesis

In his doctoral thesis, Carabelli [Car22] discusses performance-oriented communication concepts for networked control systems. He introduces and compares multiple state-dependent and opportunistic scheduling algorithms for time-critical NCS. The scheduling algorithms are evaluated using a construction with multiple simulated inverted pendulums. The control logic of the inverted pendulums uses a Kalman filter in combination with a linear quadratic regulator to calculate control instructions for the pendulum's actuator from the system's state.

In addition to simulated inverted pendulums, Carabelli also constructed a physical inverted pendulum with a 0.6 m long rod attached to a rotary encoder, measuring the angle of the rod. The rotary encoder is mounted on a 1.2 m long metal track and can be moved along the track by a stepper motor, to which it is connected with a belt. Both the rotary encoder and the stepper motor are connected to the same microcontroller. Analogously to the simulated inverted pendulums, the physical plant is also controlled with a Kalman filter and linear quadratic regulator. A picture of the inverted pendulum is shown in Figure 4.2.

To evaluate scheduling algorithms, the microcontroller sends the inverted pendulum's state to a computer via UART over USB. The computer then calculates a target speed and acceleration using a Kalman filter and linear quadratic regulator. The resulting control instruction message is then passed through a network simulation, where a to-be-evaluated scheduling algorithm is applied to the message. After the message has passed through the virtual network, the computer sends it to the microcontroller via UART over USB. The microcontroller then reads the target speed and acceleration from the message and sends matching control instructions to the stepper motor driver.

Carabelli's dissertation is of interest as the physical inverted pendulum constructed in the scope of his work forms the base for this bachelor's thesis.

3.4 Differences to this Work

While the inverted pendulum has already been extensively studied as a networked control system, this work differs from existing research.

Contrary to Zinkler [Zin16] and Nieß [Nie19], this work does not consider a simulated inverted pendulum but a physical system. This can give a more realistic experience as simulations can have idealistic assumptions which do not hold true in the physical world. For instance, a simulated inverted pendulum may experience less noise than a real one.

Additionally, not only the inverted pendulum but also the network between sensor and actuator is a real IEEE 802.3 Ethernet network, which is a difference to Carabelli's work [Car22] that connects a physical pendulum to a simulated network. Another difference to Carabelli's thesis is that the sensor and actuator logic are different physical devices. Only when the sensor and actuator are physically divided, a connection between them via a network is a realistic condition.

Besides this, Carabelli's inverted pendulum only supports a constant sampling rate for the sensor values, which results in a constant data rate. Some scheduling algorithms, like the multi-priority token-bucket scheduling considered in this work, can only show their benefits at varying data rates. Therefore, this work considers an inverted pendulum that can operate either at different constant sampling rates or dynamically adjust its sampling rate depending on the current state of the pendulum.

Other than Schindler et al. [SWVP17] and Schirle [Sch16], this work does not consider wireless networks but only wired Ethernet networks.

The most significant difference between this work and existing research is that the system constructed in this thesis is designed as a framework for arbitrary scheduling algorithms. The other studies covering physical inverted pendulums either do not consider network scheduling at all [Näg15] [Sch16] or only construct their system for one or a few fixed scheduling algorithms [SWVP17] [Car22]. Contrary to that, this work provides a framework that can be used to implement and evaluate arbitrary scheduling algorithms with a physical inverted pendulum. Additionally, a logging framework is provided, which can easily be extended to log information from arbitrary scheduling algorithms.

While Grigorjew et al. [GMH+20] propose a strict-priority multi-queue scheduling algorithm that requires all senders to comply with their contract, the multi-priority token-bucket scheduling algorithm considered in this work allows senders to violate their contract. This condition might be more suitable for applications with a dynamic data rate.

4 System Model and Problem Statement

This section will first introduce the system model that we assume in this work. Followed by this, we will formulate the problem statement for this thesis.

4.1 System Model

This section describes the properties and assumptions of the underlying system that apply to this work. The details of the inverted pendulum and the network will be explained. An overview of the system model can be seen in Figure 4.1.

From a high-level view, a rotary encoder periodically measures the current pole angle of the inverted pendulum. The sensor logic then encodes these sensor values and sends them to the actuator logic as individual Ethernet frames via an IEEE 802.3 Ethernet network with IEEE 802.1Q extensions.

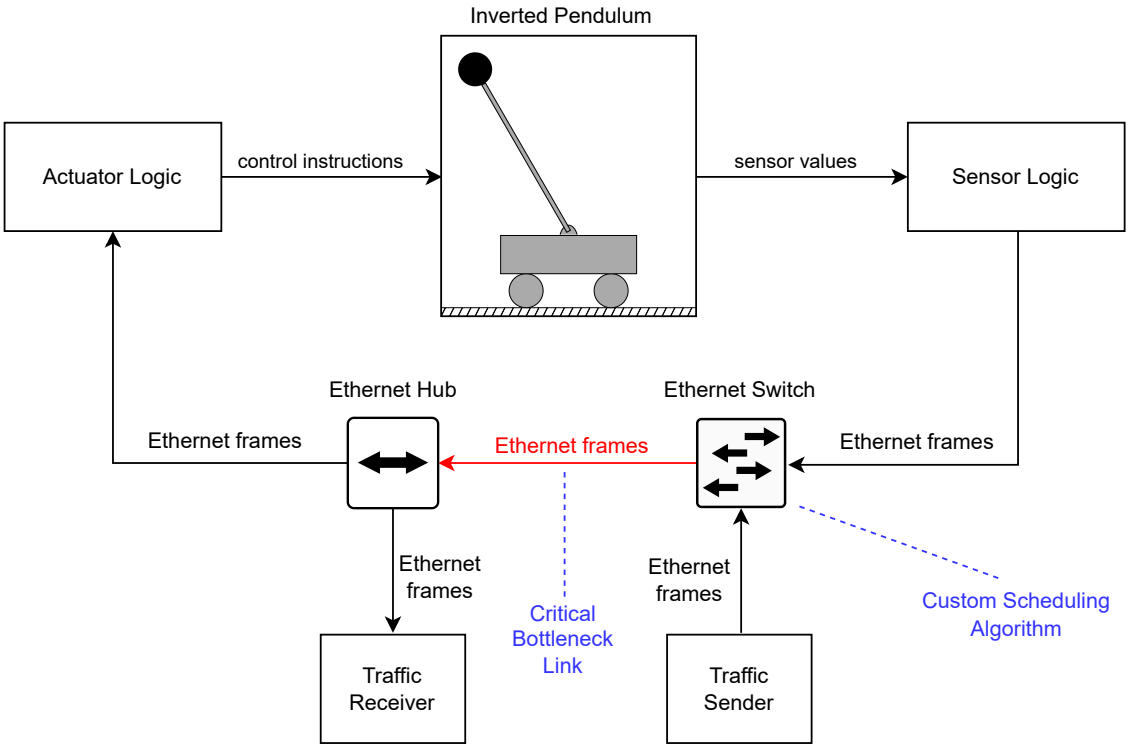


Figure 4.1: Diagram of the system model that is considered in this work. Source of the inverted pendulum graphic: [Wik12]

The actuator logic decodes the Ethernet frame and calculates control instructions for the pendulum's motor, which are then sent to the stepper motor driver of the inverted pendulum. In the following, the details about the involved network and inverted pendulum are provided.

4.1.1 Network

The network contains an Ethernet switch to which four devices are connected:

- The sensor logic of the inverted pendulum
- A cross-traffic generator
- A cross-traffic receiver
- An Ethernet hub

The switch runs an arbitrary network scheduling protocol to organize the queuing and transmission of the incoming Ethernet frames.

The aforementioned Ethernet hub is not only directly connected to the switch but also to the inverted pendulum's actuator logic and a cross-traffic receiver.

The cross-traffic generator and receiver are intended to create a load on the Ethernet switch's interface connected to the hub. As the sensor values of the inverted pendulum also get sent through this link, the connection between the switch and the hub forms a *critical bottleneck link*. This means more frames should be sent on this link than the link's bandwidth allows. The critical bottleneck link deliberately has a lower bandwidth than the other links to form a bottleneck.

4.1.2 Inverted Pendulum System

The inverted pendulum used in this work already exists and was constructed by Ben Carabelli as part of his doctoral thesis [Car22]. The pendulum consists of a cart mounted on top of a metal track with a usable range of 1.2 meters. A rotary encoder with a 0.6 m long wooden rod attached to its shaft is mounted on the cart. The rod can rotate 360° around the axis of the rotary encoder. This rotary encoder can be used to determine the angle of the rod with a resolution of 2400 steps per rotation. The cart is connected to a rubber belt, allowing the cart to be moved on the track using a stepper motor.

At the start and end of the track, limit switches are attached. These can be used to determine the track's width and the cart's position on the track at initialization. For this purpose, the controller can move the cart to the left until it hits the limit switch and then to the right until the other limit switch is activated. The number of steps performed by the stepper motor is counted during this procedure. Together with the distance that the cart moves per step of the motor ($\approx 37.4\ \mu$m per micro-step), this can be used to calculate the cart position and track length.

A picture of the inverted pendulum in the testbed can be seen in Figure 4.2.

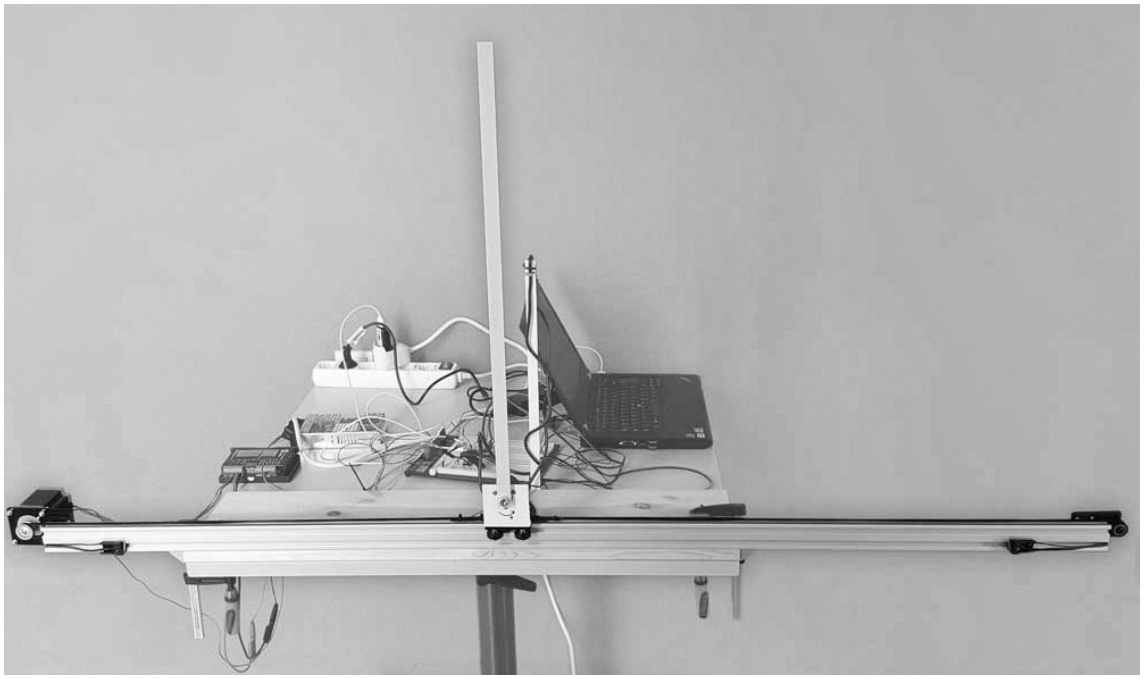


Figure 4.2: The physical inverted pendulum used in this work. Source: [Car22]

4.2 Problem Statement

In this section, the problem that this work will tackle is explained in detail.

As presented in Chapter 3, there are many approaches for packet scheduling in time-sensitive systems. A typical application for time-sensitive networking is data transmission in a networked control system. Therefore, it is of interest whether a given scheduling algorithm is capable of transmitting data in such a way that an NCS is stable. While it is possible to use a simulated network and NCS for this purpose, as demonstrated in [Zin16] and [Nie19], the evaluation of scheduling algorithms in the context of a physical NCS can give a more realistic experience. This is especially important, as networked control systems are not only a theoretical concept but are widely used in the real world, for instance, in manufacturing, automobile, aircraft, and space exploration [ZHG+20].

The inverted pendulum is a typical example of a networked control system and has extensively been studied, as discussed in Chapter 3. The main challenge for this NCS is that it is inherently unstable and therefore requires permanent low-latency communication between its sensor and its actuator. An inverted pendulum has already been constructed in previous research at the University of Stuttgart [Car22].

This work aims to construct a framework around the existing physical inverted pendulum which allows for evaluating arbitrary scheduling algorithms. This way, the performance of scheduling algorithms in the context of real-world networked control systems can be analyzed in a reproducible fashion. For this framework, the system model described in Section 4.1 is to be assumed. It should be possible to obtain different metrics about the performance of the scheduling algorithm, including

- the pendulum's state (cart position, cart velocity, pole angle, pole velocity)

- network-related performance metrics such as end-to-end latency and packet loss
- scheduling-related information (e.g., dynamic priority, token bucket state)

In particular, the framework should allow to analyze the behavior of the scheduling algorithms under a high network load generated by unrelated sources of non-time-sensitive traffic.

After the framework has been constructed, evaluations of several scheduling algorithms should be performed with it. This serves as a proof-of-concept for the validation framework and provides insights into the performance of specific scheduling algorithms. In particular, the Multi-priority Token Bucket scheduling algorithm (see Section 5.3) should be analyzed with a focus on its effectiveness and performance in time-sensitive applications. For this purpose, an implementation of MPTB for the given framework must be constructed, and fitting parameters need to be found.

5 Design

This chapter explains the design of the framework to be created within this work. First, a hardware-oriented view of the system's architecture is given. Then the used message format is discussed. Afterward, the Multi-priority Token Bucket scheduling algorithm is introduced, and the regulation of the inverted pendulum is explained. Next, an algorithm to dynamically adapt the sampling rate of the pendulum's sensor logic is explained. Followed by this, we present a method to simulate cross traffic at a switch using delays. Last, a technique to intentionally disrupt the inverted pendulum using pauses is explained.

5.1 System Architecture

In this section, the architecture of the system considered in this work will be discussed. First, the structure of the architecture and all components will be introduced. Afterward, it will be discussed why the sensor and actuator logic are structured the way they are and what alternatives exist.

5.1.1 Description of the System Architecture

The concrete architecture of the abstract system model introduced in Section 4.1 is organized as follows:

The **sensor logic** consists of two components: A Teensy 3.6 microcontroller and an ODROID C2 single-board computer. The Teensy 3.6 is directly connected to the rotary encoder of the inverted pendulum and can therefore read out the current angle of the pendulum's rod. The microcontroller transmits the sensor values to the ODROID C2 via UART over USB. The ODROID C2 receives the sensor values, encapsulates them in Ethernet frames, and sends these frames to the actuator logic via an Ethernet network.

The **actuator logic** is constructed symmetrically to the sensor logic: It consists of a Raspberry Pi CM4 single-board computer and a Teensy 3.6 microcontroller. The single-board computer receives the measurement values from the sensor logic on its Ethernet interface. The values are then unpacked and sent to the Teensy 3.6 microcontroller via UART over USB. The Teensy 3.6 then uses these sensor values to calculate control instructions for the pendulum's stepper motor. These instructions are then sent to the stepper motor driver of the inverted pendulum.

The **network** mainly consists of a 1 Gbit/s Ethernet switch and a 10 Mbit/s Ethernet Hub, which are connected via an Ethernet cable. The Ethernet switch is implemented as a software switch by an x86 Ubuntu desktop machine with a four-port 1 Gbit/s network card. A software switch was chosen over a hardware device because this allows the implementation of arbitrary scheduling algorithms

on the switch. The hub is a 10 Mbit/s EtherPrime EP-1008 nine-port Ethernet hub. The comparably low bandwidth of the hub is intentional as it acts as a bottleneck, thus requiring proper scheduling at the switch.

The sensor logic connects directly to the switch via an Ethernet cable, whereas the actuator logic is connected to the hub. Thus, Ethernet frames transmitted from the sensor to the actuator must pass the link between the switch and the hub.

Additionally, a Banana Pi M2 Berry single-board computer is connected to the switch as a **cross-traffic generator**, and a Raspberry Pi 2 is attached to the hub as a **cross-traffic receiver**. This construction is intended to create a high load on the link between the Ethernet switch and the hub, thus, making scheduling at the switch necessary.

As both the cross-traffic and the pendulum's traffic are sent from the switch to the hub, we call this part of the network the *critical bottleneck link*.

The use of the cross-traffic generator and receiver is optional. Alternatively, it is possible to create artificial latencies within the software switch to simulate a load on the critical bottleneck link (c.f. Section 5.6).

A diagram showing the hardware components of the concrete system is depicted in Figure 5.1.

5.1.2 Sensor and Actuator Structure Considerations

The sensor and actuator components both consist of a Teensy 3.6 microcontroller and a single-board computer (Raspberry Pi CM4, ODROID C2). The microcontroller communicates with the computer over USB 2.0. As an alternative, we also considered using only the microcontroller or only the single-board computer for each side. However, this has several downsides, so we decided against these options. In this section, we will explain the downsides of both alternatives in detail.

Alternative 1: Using Only the Teensy 3.6 Microcontroller

Using only the Teensy 3.6 microcontroller would imply that the Teensy 3.6 needs to handle the transmission and reception of Ethernet frames. However, the Teensy 3.6 microcontroller does not have native Ethernet support. Nonetheless, an Ethernet shield exists which communicates with the Teensy via SPI. As stated by Paul Stoffregen, Teensy's founder, the SPI intermediate step comes with a performance penalty compared to native Ethernet support, which is available in the successor Teensy 4.1 [Sto22].

Besides the performance penalty issue, buying the Teensy 3.6 Ethernet shields was difficult while this research was conducted.

Another argument against handling Ethernet on the microcontroller directly is that it is easier and more comfortable to do so on a single-board computer, as more C++ libraries are available for such a platform, and the deployment can be automated via SSH.

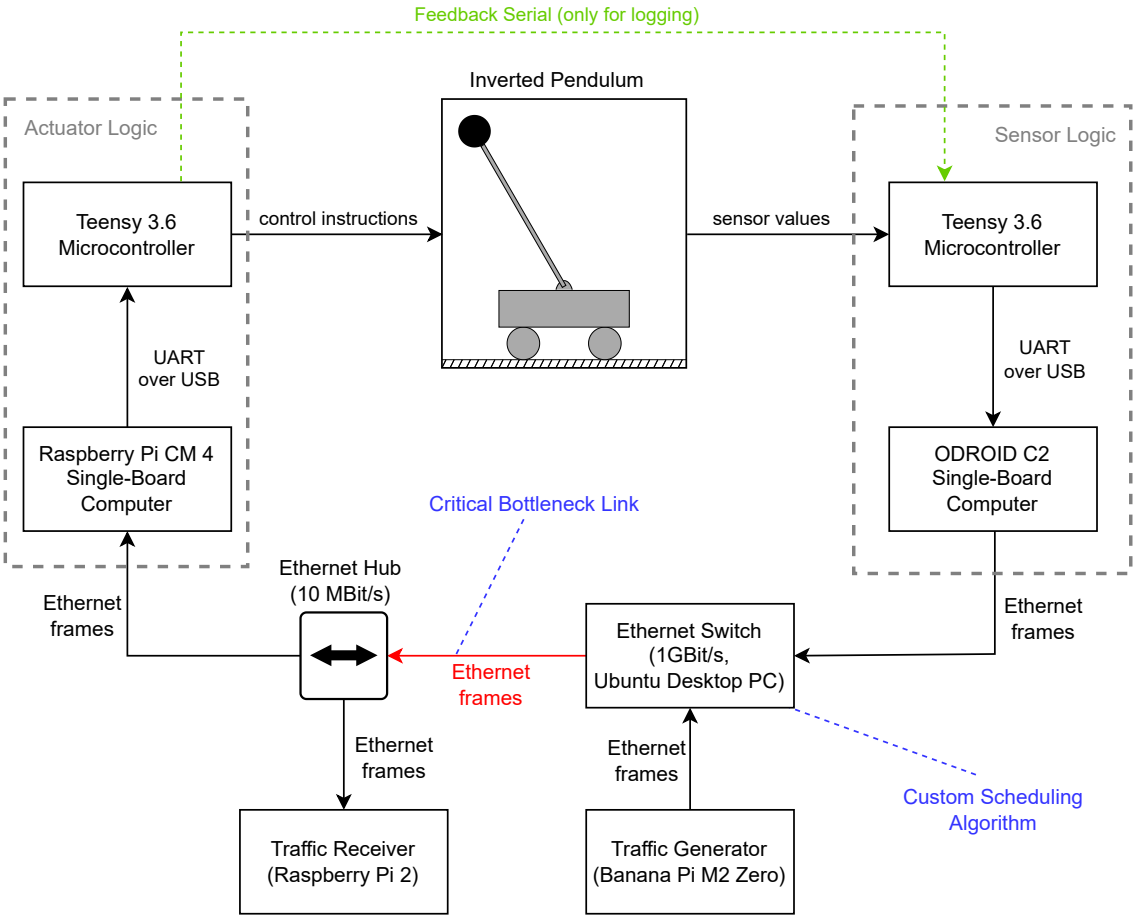


Figure 5.1: Diagram showing the hardware architecture of the system considered in this work. Compare this diagram to the system model in Figure 4.1. Source of the inverted pendulum graphic: [Wik12]

Alternative 2: Using Only the Single-Board Computer

Using only single-board computers (Raspberry Pi, ODROID) for the sensor and actuator logic comes with the issue that hardware-oriented real-time programming is more difficult on these machines. While it is possible to control the Raspberry Pi's GPIO pins from code directly, [Ras23a], the availability of libraries is much more sparse than for the Teensy 3.6.

Most significantly, the `TeensyStep` library [lun23] to control stepper motors is only available for Teensy 3.6. `TeensyStep` is a high-level library that allows to configure speeds and accelerations and handles the low-level control instructions for the stepper motor driver automatically. While stepper motor libraries exist for Raspberry Pi, they are pretty rudimentary compared to `TeensyStep`. Thus, creating something comparable to `TeensyStep` on Raspberry Pi would require significant effort.

Additionally, as single-board computers run an entire operating system, process scheduling can negatively influence time-critical operations. This is especially problematic when it comes to creating the pulses for the stepper motor, which requires accuracy in the range of microseconds.

5.2 Message Format Considerations

Different types of messages need to be exchanged between the different components of the system. The transmission of these messages happens either via the network or via UART over USB. The different message types are

- angle sensor samples (from sensor logic to actuator logic)
- pause signals (from actuator-side single-board computer to actuator-side microcontroller)
- feedback messages for logging (from actuator side to sensor side)

An easy-to-create and parse message format for these messages needs to be designed. We decided on the following string-based format:

```
MessageType:Value1;Value2;Value3;\n
```

where `MessageType`, `Value1`, `Value2`, `Value3` are placeholders for strings. The number of values is arbitrary but fixed per message type, i.e., three values is just an example.

One might argue that the use of strings is less efficient than encoding the values in a binary format, since numbers encoded as a string of decimal digits require more space than a binary number. However, strings offer the advantage of broader compatibility as they are resistant to different endiannesses.

Endianness describes how a processor organizes the bytes of numbers in the memory. Little-Endian means that the least-significant byte is stored at the lower address, followed by the more-significant bytes. In contrast, Big-Endian denotes that the most-significant byte is stored first, followed by the less-significant bytes [Moz23].

In the system considered in this work we have two processor architectures involved:

- ARM architecture (Raspberry Pi 2/4, ODROID C2, Banana Pi M2 Berry, Teensy 3.6)
- x86 (Switch desktop computer)

Both architectures use Little-Endian by default [ARM23]; thus, endianness problems cannot occur. However, as the system developed in this work is intended as a framework for other research, some components might be swapped for components with a different endianness. Thus, an endianness-resistant encoding as strings is beneficial. Other advantages of a string-based encoding include human-readability and easier parsing on arbitrary platforms.

5.3 Multi-Priority Token Bucket Scheduling

Multi-priority Token Bucket is a priority-based multi-queue scheduling algorithm based on the algorithm from Grigorjew et al. [GMH+20] but allows senders to violate their contract. In simple terms, whenever a sender violates its contract, it does not get suspended from the network but instead gets assigned a lower priority, thus leading to higher latencies. MPTB was designed at the Institute for Parallel and Distributed Systems (IPVS) of University of Stuttgart.

We consider a network consisting of one or multiple senders and receivers which communicate through a network of one or multiple switches. The switches are capable of the strict priority queueing discipline specified in the IEEE 802.1Q standard [IEE22]. The priorities are determined inside the senders, while the switches carry out the scheduling based on these priorities.

5.3.1 Underlying Scheduling Algorithm

The scheduling algorithm running on the network switches is a multi-class strict priority queueing algorithm: Let $C_0, \ldots, C_m$ be different *priority classes* where C_m is the highest priority, and C_0 is the lowest priority only providing best-effort service. Every sender is associated with a priority class. However, this association is not static but can change over time, as explained in Section 5.3.2. Therefore, the priority class is contained in every frame the sender sends.

Every priority class C_i is affiliated with a fixed per-hop *latency bound* $\delta(C_i)$. This means that every frame with priority class C_i spends at most time $\delta(C_i)$ at the current switch. δ is inversely order-preserving, meaning higher priorities are mapped to lower latencies. As C_0 only provides best-effort service, it is $\delta(C_0) = \infty$.

For every class C_i, there exists a separate *priority queue* Q_i at the switch. Whenever the switch is ready to transmit a frame, a frame from the non-empty priority queue with the largest i is selected, formally expressed as

$$C_{\text{transmit}} = C_i,\ i = \max\{i \mid Q_i \neq \emptyset, 0 \leq i \leq m\}.$$

For instance, say that Q_m is empty but $Q_0, \ldots, Q_{m-1}$ are non-empty. Then the switch picks a frame from Q_{m-1} for transmission. Note that this scheduling is *work-conserving*: Bandwidth is not exclusively reserved for individual priorities. Instead, when higher priorities do not use their available bandwidth, lower priorities can use this bandwidth.

Note that if the incoming data rates for each priority queue were not limited, this could lead to the violation of the latency bounds due to resource starvation. For example, if Q_m receives a constant incoming data stream equivalent to the switch's total egress bandwidth, the switch would pick a frame from Q_m in every iteration, as it is the highest priority. Thus, the queues $Q_0, \ldots, Q_{m-1}$ would starve and therefore violate their latency bound. To avoid latency bound violation, admission control is required, which is described in Section 5.3.3.

5.3.2 Priority Assignment

As explained above, each sending stream is dynamically assigned a priority class C_i. To determine that priority, the sender specifies a quality of service (QoS) contract at the beginning of the connection. This contract consists of the following information:

- An initial priority class C_{init}
- A burst size b
- A sustainable data rate r

b and r together specify a token bucket (c.f. Section 2.5), where r gives the sustainable data rate that the stream can send at continuously, and b gives the amount of data that can be sent at once in a burst. However, the priority determination goes beyond a simple token bucket but instead uses an extension of this concept. In simple terms, the extended token bucket is allowed to have negative bucket levels. These negative bucket levels correspond to contract violations of different severity, based on the deviation of the bucket level from zero. These contract violation severities then map to different priorities.

Definitions

First, we need to introduce some variables and mappings: Let $S_0, \ldots, S_n$ be a number of *severity levels*. Severity levels indicate how severely a sender has violated its contract. Severity level S_0 indicates that the contract is not violated, while every other S_i is a contract violation. For every severity level S_i we define a threshold $Th(S_i) \leq 0$, where $Th(S_0) = 0$ and $Th(S_1), \ldots, Th(S_n)$ can be chosen arbitrarily.

There is a mapping $Class : S \to C$, which maps severity levels to priority classes. This mapping is inversely order-preserving; that is, it holds that lower severity levels are mapped to higher priority classes.

Let $Cost : C \to \mathbb{R}_{>0}$ be a mapping that maps a priority class C_i to a cost value. This cost value indicates the cost to send a single byte with this priority. We require $Cost$ to be order-preserving, meaning lower priorities map to lower costs.

Principle of Operation:

The principle of operation will first be introduced using an informal analogy. Afterward, this will be formalized into an algorithm.

Analogy Imagine a bucket of water with several level markings on the bucket. These level markings correspond to the threshold values $Th(S_i)$, which translate to severity levels, which again map to priority classes. Thus, the amount of water inside the bucket determines the priority. If the water level is at or above the marking of $Th(S_0)$, the contract is not violated, otherwise it is. Now, if a frame should be sent, one needs to remove as much water from the bucket as it costs to send all the bytes in the frame. The new water level now determines the priority of the frame: The highest marking still below the water level is used as the Threshold $Th(S_i)$, which translates to a priority class C_j. The frame then gets sent with priority C_j.

Meanwhile, there is a continuous flow of water into the bucket with a flow rate r. However, if the water reaches the level b, all excess water overflows the bucket and can thus not be used to send frames. Figure 5.2 visualizes this concept.

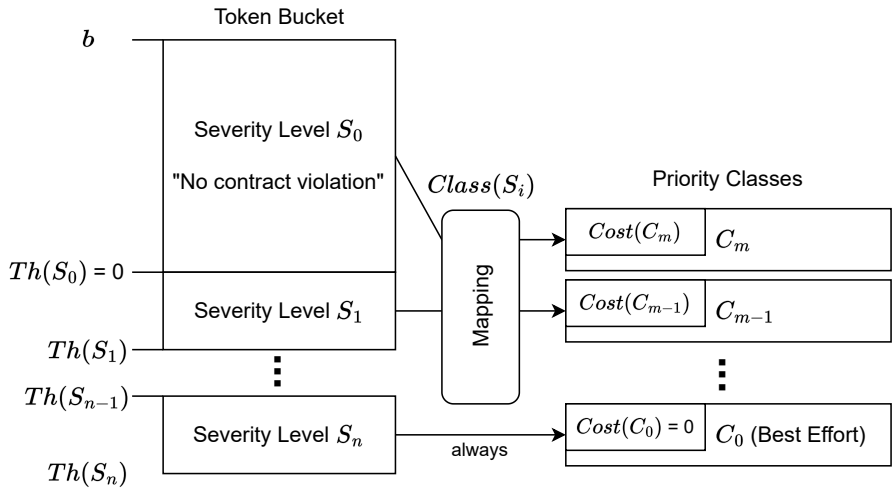


Figure 5.2: Visualization of the extended token bucket construction

Formal description The token bucket has level $B := b$ when the stream is initialized. Whenever the stream wants to send a frame of s bytes in size, it first needs to determine the priority for this frame. For this purpose, the severity level for this frame is set as the lowest severity S_i such that the bucket level is above the threshold $Th(S_i)$ after the frame has been sent with cost $Cost(Class(S_i))$. Formally, that is

$$S_i,\ i := \min\{j \mid B - Cost(Class(S_j)) \cdot s \geq Th(S_j), 0 \leq j \leq n\}$$

After S_i has been determined, the bucket level gets deducted by the frame size s bytes multiplied by the cost of one byte:

$$B := B - Cost(Class(S_i)) \cdot s$$

and the priority class C_ℓ which the frame is sent to, calculates as

$$C_\ell := Class(S_i).$$

A procedure performing these calculations can be seen in Algorithm 5.1.

5.3.3 Bandwidth Reservation and Admission Control

At every switch, each priority class C_i has a specific per-hop latency bound $\delta(C_i)$, which the switch guarantees to streams transmitting on priority C_i. To warrant this latency bound, each switch has to reserve bandwidth for every stream that uses the switch. An exception to this is the case that the stream's contract only requires best-effort service. No bandwidth needs to be reserved in this case, as there is no finite latency guarantee. If a new stream is initialized and the switch lacks enough bandwidth to fulfill the stream's contract, the stream must be rejected.

For every stream, the switch must reserve bandwidth for every severity level $S_0, \ldots, S_n$. The reservation for severity level S_i has the form of a (regular) token bucket $(b_i^{\text{res}}, r_i^{\text{res}})$. For severity level S_0 (no contract violation), these values calculate as follows:

$$b_0^{\text{res}} := b - Th(S_0), \qquad r_0^{\text{res}} := \frac{r}{Cost(Class(S_0))}$$

Algorithm 5.1 Sending a frame of size s using MPTB with current bucket level B

```
function SendFrame(frame, s, B)
    S ← S_n
    for i = 0 ... n do
        if B − Cost(Class(S_i)) · s ≥ Th(S_i) then
            S ← S_i
        end if
    end for
    C ← Class(S)
    B ← B − Cost(C)
    frame.setPriority(C)
    frame.send()
end function
```

where b, r are the token bucket values of the stream's contract. In words, r_0^{res} is selected such that the bandwidth r is reserved, and b_0^{res} is selected such that the stream can send a burst of size b without its token bucket dropping below $Th(S_0)$.

For all higher severity levels, only the *additional* bandwidth that can be sent in the lower priority classes due to lower costs, must be reserved. The reason is that strict priority queueing is a work-conserving scheduling algorithm, i.e., bandwidth not used by higher priorities is available to lower priority classes. Thus, the reservations $(b_i^{\text{res}}, r_i^{\text{res}})$ for $i > 0$ can be calculated as follows:

$$b_i^{\text{res}} := Th(S_{i-1}) - Th(S_i)$$

$$r_i^{\text{res}} := \max\left\{0,\ \max_{j<i}\left\{\frac{r}{Cost(Class(S_j))}\right\} - \frac{r}{Cost(Class(S_i))}\right\}$$

5.3.4 Using VLAN PCP to Communicate Priority

As explained above, the sender determines a priority for each frame sent. The switch then uses this priority for the strict priority queuing. For this purpose, the priority must be included in the frame.

In Section 2.6, it was explained that the IEEE 802.1Q Ethernet VLAN tag contains a three-bit priority code point (PCP) field, which can be used to communicate priorities from 0 to 7. Therefore, one can use the PCP field for priority marking as long as the switch supports at most eight priority classes.

If more priorities are required, the drop eligibility indicator (DEI) bit of the VLAN tag could also be used, allowing for up to 16 priority classes.

5.3.5 Parameter Selection

In this section, several formulas for the calculation of the parameters for MPTB will be presented. For MPTB, the following parameters need to be chosen:

- The cost $Cost(C_i)$ to send one byte for each priority class C_i
- The thresholds $Th(S_j)$ for each severity level S_j
- The stream's contract (C_{init}, b, r).

For simplicity, we assume a 1:1 mapping between severity levels S_j and priority classes C_i, i.e., the number of severity levels equals the number of priority classes, and it is $Class(S_0) = C_m, \ldots, Class(S_m) = C_0$.

We can define a maximum continuous data rate r_i of a stream for each priority class, i.e., in C_m, the sender can send at most r_m bytes/s continuously, in C_{m-1} at most r_{m-1} bytes/s and so forth. With these data rates $r_0, \ldots, r_m$ one can now calculate the costs as

$$Cost(C_i) = \frac{r}{r_i}$$

where r is the data rate from the stream's contract (C_{init}, b, r).

Now we can define a maximum per-severity burst size b_j for a given severity level S_j. This models the amount of data that can be sent at once without dropping to a worse severity level S_{j+1}. Together with the costs from the previous calculations, we can use these burst sizes to calculate the thresholds:

$$Th(S_0) := 0$$

$$\forall j \in \{1, \ldots, m\} : \qquad Th(S_j) := Th(S_{j-1}) - b_j$$

It is also possible to choose the per-severity burst sizes b_j dependent on the contract's normal burst size b. For instance, if the maximum burst of bytes should be equal in all severity levels, one can set

$$\forall j \in \{1, \ldots, m\} : \qquad Th(S_j) := Th(S_{j-1}) - Cost(Class^{-1}(S_j)) \cdot b$$

Since we consider a 1:1 mapping between severity levels and priority classes, $Class : S \to C$ is a bijective mapping, and thus, $Class^{-1}$ is well-defined.

As can be seen, with these formulas, all parameters of the MPTB scheduling algorithm can be calculated using just the sender's contract (C_{init}, b, r) as well as per-priority data rates $r_0, \ldots, r_m$.

5.4 Regulation of the Inverted Pendulum

The regulator logic of the inverted pendulum considered in this work was designed by Carabelli [Car22]. A Kalman filter is used to estimate the state of the pendulum. This state is then fed into a linear quadratic regulator (LQR) to approximate an optimal control instruction for the stepper motor driver. The details of both the filtering and the regulation are explained in this section.

5.4.1 Pre-Filtering of the Measurement Values

To remove Gaussian noise from the raw sensor values, the values are pre-filtered before they are fed into the LQR. The filtering consists of two stages: A Kalman filter as the main stage and weighted averaging as the second correction stage.

The output of the Kalman filter is calculated according to the following formula:

$$\begin{pmatrix} x'_{\text{cart},k+1} \\ v'_{\text{cart},k+1} \\ x'_{\text{pole},k+1} \\ v'_{\text{pole},k+1} \end{pmatrix} = L_x \cdot \begin{pmatrix} x_{\text{cart},k} \\ v_{\text{cart},k} \\ x_{\text{pole},k} \\ v_{\text{pole},k} \end{pmatrix} + L_u \cdot u_{\text{accel},k} + L_y \cdot \begin{pmatrix} \text{cartPos}_{k+1} \\ \text{cartSpeed}_{k+1} \\ \text{poleAngle}_{k+1} \end{pmatrix}$$

where $x_{\text{cart},k}, v_{\text{cart},k}, x_{\text{pole},k}, v_{\text{pole},k}$ are the values predicted by the Kalman filter in the previous step and $\text{cartPos}_{k+1}, \text{cartSpeed}_{k+1}, \text{poleAngle}_{k+1}$ are the new measurement values from the pendulum's angle sensor and stepper motor driver. $u_{\text{accel},k}$ is the output of the LQR from the previous step.

Thus, the output state of the Kalman filter is calculated from

- the previous state,
- the target acceleration of the previous step,
- the current measurement values.

The used matrices L_x, L_u, L_y are constant and have the following structure:

$$L_x = \begin{pmatrix} L_{x,1,1} & L_{x,1,2} & 0 & 0 \\ L_{x,2,1} & L_{x,2,2} & 0 & 0 \\ 0 & 0 & L_{x,3,3} & L_{x,3,4} \\ 0 & 0 & L_{x,4,3} & L_{x,4,4} \end{pmatrix}$$

$$L_u = \begin{pmatrix} L_{u,1} \\ L_{u,2} \\ L_{u,3} \\ L_{u,4} \end{pmatrix}$$

$$L_y = \begin{pmatrix} L_{y,1,1} & L_{y,1,2} & 0 \\ L_{y,2,1} & L_{y,2,2} & 0 \\ 0 & 0 & L_{y,3,3} \\ 0 & 0 & L_{y,4,3} \end{pmatrix}$$

The purpose of the Kalman filter is to remove Gaussian noise from the raw measurement data [Car22]. However, after the Kalman filter is applied, the values are passed through a second filtering stage:

$$\begin{aligned} x_{\text{cart},k+1} &= \alpha_{\text{kalman}} \cdot x'_{\text{cart},k+1} + (1 - \alpha_{\text{kalman}}) \cdot \text{cartPos}_{k+1} \\ v_{\text{cart},k+1} &= \alpha_{\text{kalman}} \cdot v'_{\text{cart},k+1} + (1 - \alpha_{\text{kalman}}) \cdot \text{cartSpeed}_{k+1} \\ x_{\text{pole},k+1} &= \alpha_{\text{kalman}} \cdot x'_{\text{pole},k+1} + (1 - \alpha_{\text{kalman}}) \cdot \text{poleAngle}_{k+1} \\ v_{\text{pole},k+1} &= \alpha_{\text{kalman}} \cdot v'_{\text{pole},k+1} + (1 - \alpha_{\text{kalman}}) \cdot (\text{poleAngle}_{k+1} - \text{poleAngle}_k) \cdot \frac{1}{T_s} \end{aligned}$$

where T_s is the sampling period, i.e., the time between step k and $k + 1$.

As can be seen, the output values of the Kalman filter are combined with the raw measurement values in a weighted sum. Notably, the weight α_{kalman} is usually large, e.g., Carabelli chose $\alpha_{\text{kalman}} = 0.9$ in his implementation. Thus, the Kalman filter still accounts for the significant part of the output.

The reason to include this second stage is to prevent the output values from becoming too smooth. This is problematic as sudden external disturbances to the pendulum require a quick response from the pendulum's actuator logic. However, if sudden changes are "classified" as measurement noise by the Kalman filter and thus smoothed out, this can negatively impact the actuator's response.

5.4.2 Inverted Pendulum Regulation Using LQR

A linear quadratic regulator calculates the optimal acceleration of the motor for the pendulum to stabilize. For this purpose, the output of the filtering stage explained in Section 5.4.1 gets multiplied with a constant LQR feedback gain matrix K:

$$u_{\text{accel},k+1} := \begin{pmatrix} K_{x_{\text{cart}}} \\ K_{v_{\text{cart}}} \\ K_{x_{\text{pole}}} \\ K_{\text{pole}} \end{pmatrix}^T \cdot \begin{pmatrix} x_{\text{cart},k+1} \\ v_{\text{cart},k+1} \\ x_{\text{pole},k+1} \\ v_{\text{pole},k+1} \end{pmatrix}$$

It is noteworthy that the system can not only (positively) accelerate but also slow down or change direction by giving a negative value for $u_{\text{accel},k+1}$. As the calculated acceleration might exceed the stepper motor's capabilities, the acceleration value then gets clamped to a certain range $[-a_{\max}, a_{\max}]$:

$$u^c_{\text{accel},k+1} := \max\{\min\{u_{\text{accel},k+1}, a_{\max}\}, -a_{\max}\}$$

Based on this clamped acceleration, the cart's target speed is calculated. The target speed specifies the velocity that the cart should have at the end of the next sampling period. This speed is calculated by taking into account the current speed of the cart $v_{\text{cart},k+1}$ and the clamped desired acceleration $u^c_{\text{accel},k+1}$:

$$\hat{v}_{\text{cart},k+2} := v_{\text{cart},k+1} + u^c_{\text{accel},k+1} \cdot T_s$$

where T_s is the sampling period.

Like the desired acceleration, the desired speed can exceed the stepper motor's capabilities. Thus, it also needs to be clamped to a range $[-v_{\max}, v_{\max}]$:

$$\hat{v}^c_{\text{cart},k+2} := \max\{\min\{\hat{v}_{\text{cart},k+2}, v_{\max}\}, -v_{\max}\}$$

However, as the desired velocity at the next step $k + 2$ has changed due to clamping, the desired acceleration changes too, and is now calculated from the current and desired velocity as

$$u^d_{\text{accel},k+1} := \frac{\hat{v}^c_{\text{cart},k+2} - v_{\text{cart},k+1}}{T_s}$$

The TeensyStep stepper motor library requires the acceleration and velocity to be configured as proportions of the maximum acceleration and speed. These proportions can be calculated as follows:

$$\gamma_{\text{accel}} := \frac{|u^d_{\text{accel},k+1}|}{a_{\max}}, \qquad \gamma_{\text{speed}} := \frac{\hat{v}^c_{\text{cart},k+2}}{v_{\max}}$$

Note that it is $\gamma_{\text{accel}} \in [0, 1]$ while $\gamma_{\text{speed}} \in [-1, 1]$. The reason is that the driver expects that the direction of movement is given only by the velocity and not by the acceleration.

5.5 Adaptive Sampling Rate of Inverted Pendulum

The pendulum system designed by Carabelli [Car22] is only compatible with a constant sampling rate, which was chosen to be 50 milliseconds. However, it is assumed that a dynamic sampling rate can adapt better to different situations. To be more specific, the sampling rate should be lower when the inverted pendulum is stable and should increase whenever there is a disturbance of the pendulum.

The high-level idea is that the sensor logic (c.f. Figure 4.1) monitors the pendulum's angle s_k and adapts the sampling rate accordingly. For this purpose, a compare value δ_k is calculated from the current and previous sensor values in each time step k. This δ_k is then mapped to a sampling rate. We considered three different approaches for determining δ_k based on the measured sensor values:

1. Only change from last sample value:

$$\delta_k = |s_k - s_{k-1}|$$

2. Exponentially smoothed mean over last changes

$$\delta_k = (1 - \alpha) \cdot \delta_{k-1} + \alpha \cdot |s_k - s_{k-1}|$$

   for some constant parameter $\alpha \in [0, 1]$.

3. Maximal absolute difference between the current sample and the last n samples:

$$\delta_k = \max\{|s_k - s_{k-j}| \mid j \in \{1, \ldots, n\}\}$$

Approach 1 has the disadvantage of only considering the very last change. Therefore, the sampling frequency reduces immediately after a slight change occurs. This is not desired as the pendulum might still be unstable in general, but only one change was small (e.g., when the pendulum changes direction).

Approach 2 also involves previous samples and is, therefore, better suited than Approach 1. However, the exponentially smoothed mean weights small changes equal to large changes. This means the transmission frequency does not immediately increase after one large change but only after a few samples since the previous changes might be small.

δ_k	1	2	3	4	5	6	7	8	9	≥ 10
T_k	100	90	80	70	60	50	40	30	20	10

Table 5.1: Mapping between the maximum absolute angle change δ_K over the last n samples and the sampling period T_k. Note that δ_k is in steps of the rotary encoder (2400 steps equal one rotation). T_s is in milliseconds.

Approach 3 best fits our requirements. It quickly reacts to the pendulum becoming more unstable as smaller values do not influence the maximum. It also considers previous changes, which causes the transmission frequency to not decrease immediately on a single small change of the sensor value.

However, it must be considered that Approach 3 is also the computationally most expensive approach as it requires storing the last n samples and iterating through them. Approach 1 and 2 do only require constant space and time.

We decided on Approach 3. This approach can be implemented using a ring buffer. An array stores the last n samples where the oldest value is overwritten whenever a new sample is added. The maximum change is then determined by iterating through the array and calculating the absolute difference between the current and every other sample. The maximum absolute difference is then returned.

After δ_k has been calculated, it needs to be mapped to a sampling rate. As the pendulum's sender needs to wait for time $1/\text{sampling rate}$, We will work with the sampling *period* T_k instead. Intuitively, the maximum absolute angle change δ_k should be inversely proportional to the sampling period T_k since we want to send samples more often when there are large angle changes.

The concrete mapping depends on the concrete pendulum system. In our case, the pendulum's rotary encoder has a resolution of 2400 steps per rotation, i.e., a change of 1 is equivalent to a change of $0.15°$. After some experiments with different values, we decided on the mapping shown in Table 5.1. This choice was mainly influenced by two observations: For sampling periods above 100 ms the pendulum becomes very unstable and fails quickly. On the other hand, for sampling periods below 10 ms, there was no noticeable improvement in the pendulum's stability.

5.6 Cross Traffic Simulation with Delays

As preliminary experiments have shown, it is difficult to precisely control the per-queue latency of the MPTB scheduling algorithm when using real cross traffic. This has multiple reasons:

1. In theory, the maximum per-hop latency is directly proportional to the queue size of a FIFO scheduling algorithm. However, in a real Linux system, the Qdisc's output passes through a second stage: The network card's ring buffer, which acts as a priority-independent FIFO queue. Even though this ring buffer's size can be modified using `ethtool` [Mil22], the minimum ring buffer size of the network card used in this work is 80 packets. Given an available data rate of 10 Mbit/s and a frame size of 1500 bytes, this ring buffer can introduce an additional delay of up to 96 milliseconds under full load.

2. The cross-traffic sender can have slight data rate fluctuations due to process scheduling in Linux: When the process is preempted in favor of another process running on the same machine, no packets are sent during this pause. These data rate fluctuations can lead to situations where the queue at the switch is not completely filled, thus, creating a latency that is lower than the theoretical maximum latency of the queue.

As explained in Section 5.3, MPTB guarantees a certain maximum latency bound for each queue. A given frame F sent with priority C_i experiences this maximum latency when all equal or higher priority classes $C_i, \ldots, C_m$ are under full load, that is, their queues are filled completely before F is inserted into the queue of C_i.

We can simulate this situation without filling the queues with actual frames by applying a simulated per-queue delay: Given we have priority classes $C_0, \ldots, C_m$ with their corresponding latency bounds $\delta(C_0), \ldots, \delta(C_m)$. Then, whenever a frame with priority C_i arrives at the switch, the packet is intentionally delayed by a time period of $\delta(C_i)$.

It is also possible to simulate a FIFO scheduling using this method: While FIFO does not specify a latency bound, there still is a maximum latency given by the available egress bandwidth and the maximum length of the FIFO queue. Thus, one can simulate a FIFO queue under full load by applying a constant delay δ to *all* frames processed by the switch.

Of course, it is to be noted that such deliberate delays are not desired in real-world applications. Instead, this gives a more reliable method to evaluate the performance of applications in combination with queue-based scheduling algorithms.

5.7 Intentionally Disrupting the Inverted Pendulum

The performance of scheduling algorithms for time-critical applications can be seen best in situations where low latency is crucial to stabilize the system. However, when not disrupted, the inverted pendulum system is relatively stable, even for lower sampling periods, as preliminary experiments have shown.

To put more stress on the inverted pendulum, one needs to create disruptions that negatively impact the pendulum's stability. After the pendulum has been disrupted, a quick reaction of the pendulum's actuator is required to re-stabilize the pendulum again. As the actuator logic receives the pendulum's angle via the network (c.f. Chapter 4), low-latency data transmission is crucial for fast stabilization.

Different options to disrupt the inverted pendulum have been considered:

1. A person manually gives the pendulum's rod a push.
2. A fan blows against the pendulum's rod; thus, air movement creates a drift on the rod.
3. The pendulum's motor is paused for a specific period of time. During this time, the rod will tilt either to the left or the right due to gravity and the lack of regulation by the actuator.

After some experiments, the following was found:

Option 1 creates a noticeable disruption of the pendulum. However, as a human performs this action, it is not reproducible since the push will be slightly different in each evaluation.

Option 2 does not disrupt the pendulum significantly. A 120 mm PC fan at maximum speed was used to create the disruption. At a larger distance between the fan and the rod, no significant disturbance was created. At very small distances ($\approx$ 10 cm) a noticeable disruption was created though. However, mounting the fan so close to the pendulum's rod was not feasible, as this would have blocked the rod from tilting.

Option 3 does create a noticeable disruption of the pendulum. Additionally, the disruption is reproducible as it is given only by the period of time that the pendulum is paused, which can be controlled by software. Thus, we decided on Option 3.

However, it was noticed that the significance of the disruption depends on the pendulum's state before the pause. If the rod's angle has a larger deviation from the upright position, the rod (obviously) moves more during the pause than if the rod was upright at the beginning of the pause. Thus, repeating the disruption multiple times during one evaluation is recommended, such that the impact of one outlier is less significant.

During experiments, it was found that a pause duration of 800 milliseconds works reasonably well in most cases. Larger pause durations caused the pendulum to tip over frequently, while smaller pause durations led to the pendulum not being disrupted significantly.

6 Implementation

In the following, the implementation of all components involved in the system considered in this work will be discussed in detail. An architecture diagram showing how the individual components are connected to each other is shown in Figure 5.1. First, the sensor logic consisting of a Teensy 3.6 and a single-board computer will be introduced. Next, the actuator logic, which is symmetric to the sensor logic in structure, will be discussed. Afterward, the network configuration on Linux and the switch component, including the implementation of custom scheduling algorithms, will be explained. Subsequently, the cross-traffic generator component implementation will be discussed, followed by the implementation details of the logging framework developed for this work.

6.1 Teensy Sensor Component

The Teensy sensor component was created with the Arduino platform [Ard18]. The Teensy 3.6 of the sensor logic periodically reads out the value of the pendulum's rotary encoder, which measures the pendulum's angle. The rotary encoder has a resolution of 2400 steps per rotation. Whenever a sensor value is ready for transmission, it is written to the UART over USB Serial interface together with some other data. To be specific, the following data is included in each transmitted sample:

- Raw sensor value
- Sample transmission period (in milliseconds)
- Sequence number (increments for each sample)
- Timestamp (in milliseconds since the microcontroller was powered up)

The actuator logic requires the sample transmission period as it indicates how much time has passed since the last sample. The sequence number and timestamp are used only for logging purposes to calculate the latency and packet loss. More details about the latency and packet loss calculations can be found in Section 6.8.

The aforementioned values are encoded as a string of the following structure:

```
S:sensorValue;samplingPeriodMillis;sequenceNumber;currentTime;\n
```

where the `S` indicates that this is a sample from the sensor logic. As can be seen, the values are separated by semicolons. The `\n` has the purpose of signaling a receiver where the sample ends.

The Teensy 3.6 also dynamically adapts the sample transmission period based on the absolute maximum difference of the current sample and the last 100 samples, as explained in Section 5.5. For this purpose, the microcontroller polls the rotary encoder every ten milliseconds and stores the samples in an array of size 100, acting as a rotating ring buffer. Then, the absolute maximum

difference between the most recent sensor value and the values in the ring buffer is calculated. Based on this absolute maximum difference, the new sample transmission period is determined using a lookup table.

It should be noted that the sample transmission rate is not equal to the polling rate of the rotary encoder. The encoder is read out every ten milliseconds for transmission rate adaption regardless of the transmission period. However, only after the time span given by the transmission period has exceeded, the current sensor value is transmitted.

6.2 Network Sensor Component

The network sensor component handles the transmission of the sensor values over the network. It consists of an ODROID C2 single-board computer. The program running on this machine has the following tasks:

- Receiving measurement data from the microcontroller via UART
- Determining priorities for the Ethernet frames to send
- Transmitting measurement data as Ethernet frames with the determined priority
- Logging

The software for the network sensor component is written in C++ due to performance reasons and the ability to interact with the network stack more directly. It mostly consists of two classes, the `PendulumSender` class and the abstract `PriorityDeterminer` class. Both will be discussed in the following:

6.2.1 `PendulumSender` Class

The `PendulumSender` class acts as a controller class that interacts with the microcontroller, the network, and the logger. When the class is instantiated, a UDP socket to the actuator logic gets opened. In addition, the program binds to the UART serial port to which the Teensy 3.6 microcontroller is connected. Afterward, until manually stopped, the program constantly checks the serial port for new input. As mentioned in Section 6.1, the input is always encoded as a string ending with a newline character (`\n`). It is first checked whenever new data is received, whether it is a sensor value or a feedback value (c.f. Section 6.8). If it is a feedback value, the value gets sent to the logger.

Otherwise, in the case of a sensor value, the value is first padded to a string of 32 bytes in length using "`#`" characters. This creates a payload of constant length, such that the data rate stays constant for a constant sample transmission rate. The pure string encoding without padding does not guarantee that since the string encoding of numbers varies in length depending on the number of decimal places.

Afterward, the Priority Determiner's `reportPacketReadyToSend` method gets called with the payload length of 32 bytes, such that the priority determiner can take this amount of data into account when calculating the priority. Then the priority for the current packet is obtained from the priority

Listing 6.1 `PriorityDeterminer` class

```
class PriorityDeterminer {
  public:
  virtual unsigned int getPriority() = 0;
  virtual void reportPacketReadyToSend(int payloadSizeBytes) = 0;
  virtual SchedulingInfoEntry* getSchedulingInfoEntry() = 0;
  virtual std::string getDebugInfoString() = 0;
  virtual void resetState() = 0;
  ...
};
```

determiner. Next, the UDP socket's `SO_PRIORITY` field gets set this priority, and the padded payload gets sent to the actuator logic via the socket. Additionally, the payload and some other statistics are passed to the logger. The code performing said tasks is shown in Listing 6.2.

Remark: The `SO_PRIORITY` field determines the `skb->priority` field of the local egress queue. Due to the network configuration described in Section 6.5, the `skb->priority` value is then included in the resulting Ethernet frame as the PCP value. Hence, the `SO_PRIORITY` field controls the PCP value of the VLAN tag.

6.2.2 `PriorityDeterminer` Superclass

One goal of this work is to create a framework for the evaluation of arbitrary scheduling algorithms. For this purpose, an abstract `PriorityDeterminer` class is provided (see Listing 6.1). This class, most importantly, offers two methods:

- `void reportPacketReadyToSend(int payloadSizeBytes)`: Whenever a packet is ready for transmission, the sender should call this method, such that the priority determiner knows how much data is sent.
- `unsigned int getPriority()`: This method returns a priority (0-7), which can then be included in the packet as a VLAN PCP tag by the sender.

Classes can inherit `PriorityDeterminer` to implement arbitrary algorithms to determine priorities. For instance, we implemented the following approaches as subclasses of this class:

- Constant Priority
- Token Bucket
- Multi-Priority Token Bucket

Therefore, modifying the sender side for arbitrary priority-based scheduling algorithms is just as easy as creating a subclass of `PriorityDeterminer`, creating an instance of it, and injecting it into the `PendulumSender` instance at object creation.

Listing 6.2 Transmission of a measurement value as a UDP packet

```
void PendulumSender::sendPacket(std::string payload) {
  // Pad payload width '#' to 32 bytes
  std::string paddedPayload = payload;
  paddedPayload.append(32 - payload.size(), '#');

  priorityDeterminer->reportPacketReadyToSend(paddedPayload.size());
  int priority = priorityDeterminer->getPriority();
  senderSocket.set_option(SOL_SOCKET, SO_PRIORITY, priority);
  senderSocket.send_to(paddedPayload, receiverAddress);
  packetCount++;
  bytesSentTotal += paddedPayload.size();

  logger.log(packetCount, bytesSentTotal, payload,
  priorityDeterminer->getSchedulingInfoEntry());
  ...
}
```

6.3 Network Actuator Component

The network actuator component consists of a Raspberry Pi CM4 single-board computer. The software running on this board is written in C++ and is part of the same code base as the network sensor component described in Section 6.2. The program has the following tasks:

- Receiving measurement data from the network
- Transmitting the measurement values to the actuator-side microcontroller via UART
- Sending pause signals to the actuator-side microcontroller
- Logging of network-received payloads and pendulum-related data from the microcontroller

When the class is instantiated, it binds to a UDP socket at port 3000 such that incoming packets can be received. Besides this, the UART serial port to the actuator-side Teensy 3.6 is opened. Now, until manually stopped, the program repeatedly blocks until it receives a new UDP packet. When such a packet is received, the padding (c.f. Section 6.2) gets stripped from its payload. Next, the un-padded payload gets written to the UART serial port of the actuator-side microcontroller. Additionally, the payload gets passed to the logger, among other statistics.

Besides this, the program sends pause signals to the microcontroller via UART in periodic intervals. Pause signals, as explained in Section 5.7, instruct the actuator of the pendulum to stop the stepper motor for a specified amount of time. The pause signals are encoded as strings of the following form:

`PAUSE:pauseDurationMillis;\n`

where `pauseDurationMillis` is a placeholder for an integer, stating for how long the actuator should pause.

The component also reads statistics from the actuator-side microcontroller via the UART serial interface and passes them to the logger. As explained in more detail in Section 6.8, these statistics include the current cart position, cart speed, and acceleration, among others.

6.4 Teensy Actuator Component

The Teensy actuator component's code is a modified version of the Arduino sketch written by Ben Carabelli during the original construction of the pendulum [Car22]. Carabelli's version of the code was only intended for local data processing, i.e., the rotary encoder and the stepper motor driver are connected to the same microcontroller, and no data is sent over a real network. Additionally, the original code can only handle a constant sampling period of 50 milliseconds. The project also does not feature an option to pause the motor for a given period of time, as described in Section 5.7. Therefore, in total, we made the following modifications to the code:

- Remove rotary encoder read-out (not part of the actuator logic)
- Add capability to receive sensor values via UART over USB
- Add support for an adaptive sampling period
- Add support for a pause command which pauses the motor for *n* milliseconds

In the following, the features of the now modified code for the actuator-side microcontroller will be explained in detail.

6.4.1 Receiving Samples via UART Over USB

In contrast to Carabelli's original implementation, samples are not obtained from a local rotary encoder but instead can be received via the UART over USB interface of the Teensy 3.6. For this purpose, in the main loop of the sketch, it is checked in every iteration whether a new sample has been received via UART.

As already mentioned in Section 6.1, the samples are strings of the following form:

```
S:sensorValue;samplingPeriodMillis;sequenceNumber;currentTime;\n
```

For instance, a sample could look like this:

```
S:-1204;50;1234;62345234;\n
```

Besides samples, pause signals can also be received over the same UART interface, as explained in Section 6.4.3. The program distinguishes these values by their prefix: Samples start with "`S:`" while pause signals start with "`PAUSE:`".

Whenever a regular sample is received, the sensor value gets stored in a global variable `currentEncoderValue`, and the `newEncoderValueAvailable` is set to `true` such that in the next iteration of the main loop, the motor control instructions get updated based on the new sample. Additionally, the controller parameters get updated based on the new sampling period, as described in Section 6.4.2, and a feedback message is transmitted to the sensor logic as described in Section 6.8.

6.4.2 Handling of Adaptive Sampling Period

As explained in Section 6.4.1, every sample also includes a sampling rate value, which indicates the time that has passed since the previous sample was transmitted. When the sampling rate changes, the parameters for the regulator logic, namely the matrices L_x, L_u, and L_y for the Kalman filter as well as the LQR feedback gain matrix K need to be adapted. For more information about the mentioned matrices see Section 5.4.

To adapt the parameters, the `updateParametersForSamplingPeriod` method is called whenever a new sample is received. This method contains the correct parameters for a sampling period of 10, 20, 30, . . . , 130, 140, 150, 175, 200, and 300 milliseconds as pre-calculated values. The beginning of this function can be seen in Listing 6.3. The function uses a switch-case construction to map the sampling period values to the corresponding parameters.

The parameters themselves were pre-calculating using a script written in the Julia programming language. This script was created by Carabelli as part of his doctoral thesis [Car22] and will not be discussed further in this work.

6.4.3 Pause Feature

As explained in Section 6.3, the network actuator component can send a command of the form

```
PAUSE:pauseDurationMillis;\n
```

via the UART over USB interface. Here, `pauseDurationMillis` is a placeholder for a positive integer. The microcontroller listens for such commands and pauses the motor for the provided amount of milliseconds. This is achieved by setting the motor speed to 0 using the `TeensyStep` library.

During the pause, incoming samples are still received via the UART over USB interface, such that the feedback mechanism described in Section 6.8 does not report packet loss due to discarded samples.

6.4.4 Regulator

The regulator's task is to calculate control instructions for the stepper motor driver based on sensor values, such that the inverted pendulum's rod is kept upwards. As input values, the following metrics are available in every time step:

1. Position of the cart on the track in meters, where 0 is the center of the track (cartPos_{k+1})
2. Velocity of the cart in m/s (cartSpeed_{k+1})
3. Angle of the pole in degrees (poleAngle_{k+1})

It is noteworthy that the cart position and velocity are not actual sensor values but are instead based on the history of the stepper motor's movements. This is facilitated by the `TeensyStep` stepper motor library [lun23], which is not only used to control the motor but also provides such metrics. However,

Listing 6.3 Modification of the regulator parameters based on the sampling period

```
void updateParametersForSamplingPeriod(unsigned samplingPeriod){
  if(samplingPeriod == balancePeriod){
    return; // Nothing to do; parameters already set correctly.
  }
  balancePeriod = samplingPeriod;
  Ts = balancePeriod / 1000.0;  // sampling period in seconds
  fs = 1000.0 / balancePeriod;  // sampling frequency in Hz

  switch(samplingPeriod){
    case 10:
    // Matrix K for LQR:
    Kxc = 5.460879579024502;
    Kvc = 6.317330404682753;
    Kxp = -45.38283069547128;
    Kvp = -12.003680491201385;

    // Matrices L_x, L_u, L_y for Kalman filter:
    Lx11 = 0.77284936269;
    Lx21 = -0.003863314472;
    Lx12 = 0.00159557575;
    Lx22 = 0.773001026397;
    Lx33 = 0.486684863435;
    Lx43 = -2.298789201109;
    Lx34 = 0.004864576672;
    Lx44 = 0.976322619553;
    Lu1 = -2.2686711e-5;
    Lu2 = 0.00773020343;
    Lu3 = 3.4742919e-5;
    Lu4 = 0.014114941957;
    Ly11 = 0.22715063731;
    Ly21 = 0.003863314472;
    Ly12 = 0.006132917877;
    Ly22 = 0.226960340458;
    Ly33 = 0.513655922912;
    Ly43 = 2.437239843769;
    break;

    case 20:
    ...
  }
}
```

these values would deviate from the truth if there was some slippage in the connection between the motor and the rubber belt to which the cart is attached. Own experiments have shown, however, that such slippage does not occur.

Other than the cart position, the angle of the pole is a real sensor value obtained by the sensor logic and then sent through the network to the actuator logic. Thus, there is a delay between the value's measurement and the value's arrival at the actuator logic due to network latency.

Whenever a new pole angle sensor value arrives at the actuator-side microcontroller, the microcontroller also obtains the cart position and speed, and feeds these values into the regulator logic. The regulator logic first pre-processes the measurement values using a Kalman filter and a second anti-oversmoothing stage. This results in a state vector of four values: Cart position, cart speed, pole position, and pole speed. This state is then fed into a linear quadratic regulator, which calculates the desired acceleration of the stepper motor (in m/s^2). This acceleration is then clamped to the stepper motor's maximum acceleration, and the desired speed at the next time point is calculated from the current speed and the clamped acceleration. A detailed mathematical description of these calculations can be found in Section 5.4.

The calculated acceleration and speed are then passed to the `TeensyStep` library, which drives the motor accordingly. A code excerpt showing how the described regulator logic is implemented can be found in Listing 6.4.

6.5 Network Configuration

For the network considered in this work to function, some advanced configuration is required on all participating devices. In this section, we will explain in detail how the switch PC and the endpoints need to be configured.

6.5.1 Configuration of the Switch PC

The network switch running a custom scheduling algorithm is implemented using a desktop computer with a 4-port Ethernet card. As an operating system for this PC we selected Ubuntu 22.04. For the computer to act like a switch, some configuration is required. The configuration is accomplished using the `ip` command line tool, which is part of Ubuntu. In the following, We will explain the configuration of the switch PC in detail. The corresponding configuration bash script can be found in Listing 6.5. To better understand the script, it is also helpful to look at the interface names in the network topology in Figure 6.1.

For every involved physical interface of the network card, the configuration script sets up a virtual VLAN interface with VLAN ID 100. For instance, for the interface `enp1s0f0`, the virtual interface `enp1s0f0.100` gets created. These virtual interfaces are then associated with a so-called `ingress-qos-map` and `egress-qos-map`. This is a mapping between the PCP (priority code point) value of the VLAN tag (c.f. Section 2.6) and the `skb->priority` value used for the scheduling:

- `ingress-qos-map` maps the PCP of an *incoming* Ethernet frame to the `skb->priority` field.

Listing 6.4 Code excerpt of the regulator logic running on the actuator-side microcontroller

```
if (newEncoderValueAvailable) {
  ...
  float cartPos = METER_PER_MSTEP * motor.getPosition();
  float cartSpeed = getCartSpeedMeter();
  float poleAngle = getPoleAngleRad();
  float x_cart_p = x_cart; v_cart_p = v_cart; x_pole_p = x_pole; v_pole_p = v_pole;

  // Kalman filter:
  x_cart = Lx11*x_cart_p + Lx12*v_cart_p + Lu1*u_accel + Ly11*cartPos + Ly12*cartSpeed;
  v_cart = Lx21*x_cart_p + Lx22*v_cart_p + Lu2*u_accel + Ly21*cartPos + Ly22*cartSpeed;
  x_pole = Lx33*x_pole_p + Lx34*v_pole_p + Lu3*u_accel + Ly33*poleAngle;
  v_pole = Lx43*x_pole_p + Lx44*v_pole_p + Lu4*u_accel + Ly43*poleAngle;

  // Anti-oversmoothing:
  x_cart = alpha * x_cart + (1 - alpha) * cartPos;
  v_cart = alpha * v_cart + (1 - alpha) * cartSpeed;
  x_pole = alpha * x_pole + (1 - alpha) * poleAngle;
  v_pole = alpha * v_pole + (1 - alpha) * (poleAngle - poleAngle_p) * (1/Ts);
  poleAngle_p = poleAngle;

  // Controller:
  u_accel = Kxc * x_cart + Kvc * v_cart + Kxp * x_pole + Kvp * v_pole; // Apply LQR
  u_accel = constrain(u_accel, -aMaxMeters, aMaxMeters); // clamp to range
  float target_speed = cartSpeed + u_accel * Ts; // [m/s]
  target_speed = constrain(target_speed, -vMaxMeters, vMaxMeters); // clamp to range
  u_accel = (target_speed - cartSpeed) * fs; // correct acceleration for speed clamping
  float accel_factor = std::abs(u_accel) / aMaxMeters; // (factor must be positive)
  float speed_factor = target_speed / vMaxMeters;

  // Pass values to TeensyStep library:
  rotate.overrideAcceleration(accel_factor);
  rotate.overrideSpeed(speed_factor);
  ...
}
```

- `egress-qos-map` maps the `skb->priority` field of an *outgoing* packet to the PCP of the Ethernet frame, i.e., the VLAN tag of this frame now contains the `skb->priority` value.

Thus, in total, a frame faces two mappings when being processed by the switch: The mapping from the PCP to `skb->priority` when the frame arrives at the switch and the mapping from `skb->priority` to PCP when the frame gets transmitted. The PCP must get mapped to the `skb->priority` field since the scheduling algorithm (Qdisc) only sees the `skb` data structure but not the contents of the incoming Ethernet frame. Thus, the PCP itself is not preserved.

To connect the different interfaces of the switch to one another, the configuration script creates a virtual bridge to which all involved VLAN interfaces are connected. To ensure that the frames passed through the switch are actually processed by the kernel, the promiscuous mode is enabled for all involved interfaces. Otherwise, frames with non-local destinations would be dropped [Arc23].

An optional but useful configuration step is to assign an IP address to the switch PC. Technically, the switch does not require an IP address, as it operates on the data link layer. However, when an IP address is assigned, the switch PC can also be used to access the other machines in the network via SSH, which makes administration easier.

6.5.2 Configuration of the Endpoint Hosts

Some configuration is also required on all endpoint devices, namely the sensor logic, the actuator logic, and the cross-traffic sender and receiver. A Bash script setting up the required configuration can be seen in Listing 6.6.

Most importantly, on the endpoints, it is also necessary to create a VLAN interface `eth0.100` based on the physical interface `eth0`. Otherwise, the VLAN header, including the PCP (priority code point), would not be available. It is also essential to define an `egress-qos-map` which maps the `skb-priority` value to a PCP, which is then included in the Ethernet frame.

Theoretically, one could also omit the `egress-qos-map` and instead set the PCP of the Ethernet frame directly from the sender code. This, however, has two significant disadvantages, as own investigations have shown:

- This only works with raw Ethernet sockets but not with UDP/IP sockets. The use of raw Ethernet sockets is impractical for most real-world applications.
- Raw Ethernet sockets require root privileges on Linux, which is usually undesirable in real-world applications. (However, when PCP value 7 should be used, the program requires root privileges for UDP sockets too.)

Instead, with an `egress-qos-map` in place, a program can simply open a UDP or TCP socket and set the desired priority value using the `SO_PRIORITY` socket option. This socket option controls the `skb->priority` field within the local network stack, which is then included as the PCP field in the resulting Ethernet frame.

Another configuration required on the endpoint is the assignment of an IP address. This is necessary since the endpoints communicate using UDP/IP sockets. Thus, every endpoint needs a distinct IP address. The IP address is also useful as it allows to access the endpoints via SSH.

6.6 Switch Component: Custom Scheduling

A key goal of this work is to provide an easy way to implement custom scheduling algorithms on Linux-based software switches. In this section, we will explain in detail how Linux can be extended by arbitrary network scheduling algorithms.

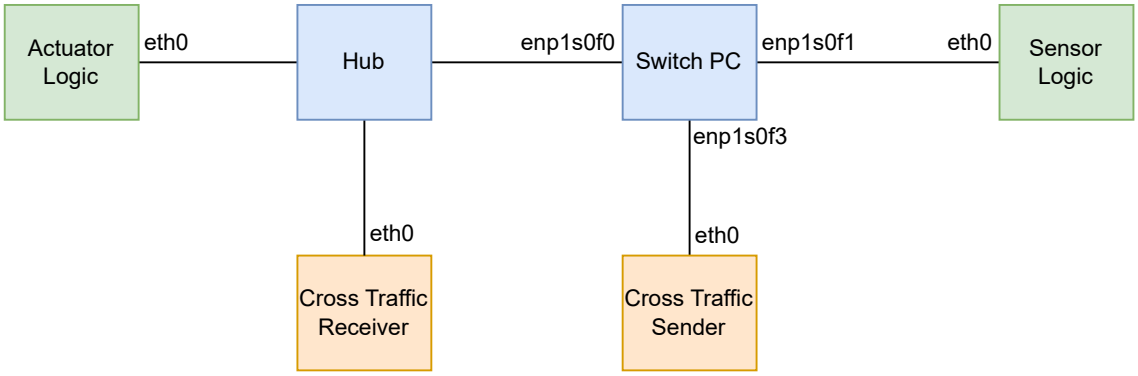


Figure 6.1: Network topology of the system considered in this work, including network interface names

Listing 6.5 Network configuration script for the software switch

```
# Setup interfaces for VLANs (on VLAN ID 100):
sudo ip link add link enp1s0f0 name enp1s0f0.100 type vlan id 100 \
  ingress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7 \
  egress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
sudo ip link add link enp1s0f1 name enp1s0f1.100 type vlan id 100 \
  ingress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7 \
  egress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
sudo ip link add link enp1s0f3 name enp1s0f3.100 type vlan id 100 \
  ingress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7 \
  egress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7

# Create bridge:
sudo ip link add name br0 type bridge
sudo ip link set dev enp1s0f0.100 master br0
sudo ip link set dev enp1s0f1.100 master br0
sudo ip link set dev enp1s0f3.100 master br0

# Enable promiscious mode:
sudo ip link set dev enp1s0f0.100 promisc on
sudo ip link set dev enp1s0f1.100 promisc on
sudo ip link set dev enp1s0f3.100 promisc on

# Turn on all involved interfaces:
sudo ip link set dev enp1s0f0.100 up
sudo ip link set dev enp1s0f1.100 up
sudo ip link set dev enp1s0f3.100 up
sudo ip link set dev enp1s0f0 up
sudo ip link set dev enp1s0f1 up
sudo ip link set dev enp1s0f3 up
sudo ip link set dev br0 up

# Set IP address of switch pc:
sudo ip address add 10.0.1.4/24 dev br0
```

Listing 6.6 Network configuration script for an endpoint node

```
# Add VLAN (with VLAN ID 100)
sudo ip link add link eth0 name eth0.100 type vlan id 100 \
  ingress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7 \
  egress-qos-map 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7

# Enable all involved interfaces
sudo ip link set dev eth0 up
sudo ip link set dev eth0.100 up

# Set IP address (different for each endpoint, but same IP subnet)
sudo ip address add 10.0.1.1/24 dev eth0.100
```

6.6.1 Implementing the Scheduling Algorithm

As explained in Section 2.7, the network scheduling in Linux is based on Qdiscs (queueing disciplines). There are several Qdiscs available in Linux by default, e.g., `prio` for priority scheduling, `fifo` for first-in-first-out scheduling. These Qdiscs are implemented in `/net/sched/sch_*.c` files within the kernel source [Lin23].

Thus, if one wants to implement a custom scheduling algorithm, one needs to create a new Qdisc as a `sch_*.c` file. In this file, an instance of the `Qdisc_ops` struct (shown in Listing 2.1) must be created. This involves the implementation of all the functions listed in the struct. Most importantly, the `enqueue` and `dequeue` methods must be implemented. The kernel calls these methods to pass frames to the Qdisc or get the next frame for transmission, respectively. Hence, any scheduling algorithm can be realized by implementing `enqueue` and `dequeue` appropriately. `Qdisc_ops` also includes an `id` field which stores a string. This field should store the (unique) name of the Qdisc and is used by `tc` to identify it.

As the Qdisc is used as a Linux kernel module, it must also provide functions that should be called on the insertion and removal of the module. These functions should internally call the `register_qdisc` and `unregister_qdisc` functions which are provided by the kernel in `/include/net/pkt_sched.h`. These functions make the Qdisc module available to the rest of the kernel or remove it from the kernel, respectively. The init and exit functions are then passed to the kernel using the `module_init` and `module_exit` kernel functions, which are part of the kernel module framework. A skeleton for the implementation of a custom Qdisc is shown in Listing 6.7.

There are two options to make the custom Qdisc available to the kernel:

1. Inserting the created `sch_*.c` file into the `/net/sched` directory of the Linux kernel source and then re-compiling and installing the modified kernel.
2. Compiling `sch_*.c` to a kernel module and then adding this module to the kernel.

Option 1 has the disadvantage that re-compiling the entire kernel is time-intensive. Own experiments have shown that the initial compilation time heavily depends on the CPU: Compile times between six minutes (AMD Ryzen 7 2700X, eight cores) and three hours (VirtualBox virtual CPU with

Listing 6.7 Implementation of a custom Qdisc called `myqdisc`

```
// Includes (missing)
// ...

static int myqdisc_enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **to_free){
  // Custom Implementation
}

static struct sk_buff *myqdisc_dequeue(struct Qdisc *sch){
  // Custom Implementation
}

// Implementation of all other functions (missing)
// ...

static struct Qdisc_ops myqdisc_qdisc_ops __read_mostly = {
  .next   = NULL,
  .cl_ops   = &myqdisc_class_ops,
  .id   = "myqdisc",
  .priv_size  = sizeof(struct myqdisc_sched_data),
  .enqueue  = myqdisc_enqueue,
  .dequeue  = myqdisc_dequeue,
  .peek   = myqdisc_peek,
  .init   = myqdisc_init,
  .reset    = myqdisc_reset,
  .destroy  = myqdisc_destroy,
  .change   = myqdisc_tune,
  .dump   = myqdisc_dump,
  .owner    = THIS_MODULE,
};

static int __init myqdisc_module_init(void){ // Called when kernel module inserted
  return register_qdisc(&myqdisc_qdisc_ops);
}

static void __exit myqdisc_module_exit(void){ // Called when kernel module removed
  unregister_qdisc(&myqdisc_qdisc_ops);
}

// Register init and exit functions at kernel:
module_init(myqdisc_module_init)
module_exit(myqdisc_module_exit)
```

one core) have been measured. However, subsequent compilations are significantly faster as the build-process re-uses large parts of the original build results. Another disadvantage of Option 1 is that a system reboot is required after every change to the source.

Option 2 has remarkable advantages over Option 1 since the compilation process is significantly faster (in the range of seconds), and the kernel module can be added to and removed from the kernel at runtime without rebooting. Therefore, we decided on Option 2.

To facilitate the process of compiling and (re-) inserting the kernel module into the kernel, we created a Bash script that

1. Compiles the Qdisc-implementing `sch_*.c` file to a `sch_*.ko` kernel object file.
2. Deletes the custom Qdisc from network interfaces if applicable.
3. Removes the module from the kernel using the `rmmod` command if already inserted
4. Inserts the new `sch_*.ko` file into the kernel using the `insmod` command.

6.6.2 Exposing the Qdisc via `tc`

As explained in Section 2.7, the `tc` command line tool is used to configure the scheduling algorithm of a network interface. For this to work, a binding needs to be created, which makes the Qdisc available to `tc`. Thus, when creating a custom Qdisc, one also needs to create a matching Qdisc binding.

Such a binding is a `q_*.c` file written in the C programming language, consisting of three methods:

- `explain`: Prints a string informing the user about the command line arguments that are available for this Qdisc. This string is printed when the user enters invalid command line arguments.
- `parse_qopt`: This method parses Qdisc-specific arguments from the command line and passes them to the Qdisc. For instance, the `prio` Qdisc accepts the `bands` argument, which sets the number of priority queues. If the custom Qdisc does not need any user-configurable parameters, the implementation of this method can be omitted.
- `print_qopt`: This method prints configuration information about a concrete Qdisc instance bound to an interface. This information is displayed to the user when using the `tc qdisc show` command. For example, for the `prio` Qdisc, this method displays the number of queues ("bands") as well as the "priomap", which is the mapping of the `skb->priority` field to the priority queues. If such information is not of interest, the implementation of this method can also be omitted.

The `parse_qopt` and `print_qopt` are bundled into a `qdisc_util` struct together with an `id` field. This `id` field is important as it links the actual Qdisc inside the kernel to `tc`. A skeleton for the implementation of a binding for `tc` can be seen in Listing 6.8.

To add the binding to `tc`, it is necessary to re-compile `tc`. As `tc` is part of the `iproute2` package, one needs to download the `iproute2` source [The22] and then insert the newly created `q_*.c` file into the `iproute/tc` directory. Then, `tc` can be compiled into an executable. This executable should then be used instead of the system-wide `tc` command.

Listing 6.8 `tc` binding of a custom Qdisc called `myqdisc`

```
// Includes (missing)
// ...

static void explain(void){
  fprintf(stderr, "Usage: ...");
}

static int myqdisc_parse_opt(struct qdisc_util *qu, int argc, char **argv, struct nlmsghdr *n,
 const char *dev){
  // Custom code to parse command line arguments
}

int myqdisc_print_opt(struct qdisc_util *qu, FILE *f, struct rtattr *opt){
  // Custom code to print out status information about the Qdisc
}

struct qdisc_util myqdisc_qdisc_util = {
  .id = "myqdisc", // Must be idential to Qdisc_ops->id in Qdisc implementation
  .parse_qopt = myqdisc_parse_opt,
  .print_qopt = myqdisc_print_opt,
};
```

It is now possible to configure the custom Qdisc named `myqdisc` for an interface `eth0` using the following command:

```
sudo ./iproute2/tc/tc qdisc add dev eth0 root myqdisc
```

6.6.3 Applying Simulated Delays to Queue Outputs

As explained in Section 5.6, cross traffic can be simulated by applying delays to all frames handled by specific queues. To implement this, one could, in theory, implement the corresponding logic in the source code of the Qdisc. However, as Linux allows to combine multiple Qdiscs in a tree-like structure (c.f. Section 2.7.2), there is a much simpler solution: The `netem` Qdisc [LP22] is a special queueing discipline that allows to simulate different network conditions. Most importantly, this Qdisc can be used to generate artificial delays. Thus, one can attach a `netem` Qdisc to each output queue of the custom Qdisc to generate a deterministic per-priority latency. An example showing how to set up strict priority queueing with simulated per-queue delays is shown in Listing 6.9.

6.6.4 Configuring the Queue Size

The queue size has an impact on the per-hop latency of the switch. Therefore, we need a way to configure it. Most existing Qdiscs in Linux use the `txqlen` attribute of the interface as their queue length in frames. This value can be set using the `ip` tool. For example, to set the queue length of interface `eth0` to 50 frames, the following command can be used:

```
sudo ip link set dev eth0 txqlen 50
```

Listing 6.9 Configuration of a priority scheduling with eight priorities and a simulated delay for each priority queue. Note that, contrary to the notation in this paper, lower numbers correspond to higher priorities.

```
# Set up priority scheduling with 8 priority classes:
sudo tc qdisc add dev eth0 root handle 1 prio bands 8 priomap 0 1 2 3 4 5 6 7 7 7 7 7 7 7 7 7

# Add an individual delay to each priority class using netem:
sudo tc qdisc add dev eth0 parent 1:1 handle 21 netem delay 5ms
sudo tc qdisc add dev eth0 parent 1:2 handle 22 netem delay 10ms
sudo tc qdisc add dev eth0 parent 1:3 handle 23 netem delay 15ms
sudo tc qdisc add dev eth0 parent 1:4 handle 24 netem delay 20ms
sudo tc qdisc add dev eth0 parent 1:5 handle 25 netem delay 25ms
sudo tc qdisc add dev eth0 parent 1:6 handle 26 netem delay 30ms
sudo tc qdisc add dev eth0 parent 1:7 handle 27 netem delay 35ms
sudo tc qdisc add dev eth0 parent 1:8 handle 28 netem delay 40ms
```

The queue of the scheduling algorithm is not the only data structure buffering frames. The interface itself also has two ring buffers: One to store received frames before they are processed by the kernel (`rx`) and one to store frames to be transmitted after they leave the scheduling algorithm (`tx`). Unfortunately, the size of this ring buffer can only be changed within certain limits. For instance, the computer used as a software switch in this work only allows a minimum ring buffer size of 80 frames. This value can be configured via `ethtool` for both the receiving and the transmitting ring buffers. For instance, when both ring buffers of the interface `eth0` should be set to 80 frames, the following commands need to be run:

```
sudo ethtool -G eth0 rx 80

sudo ethtool -G eth0 tx 80
```

6.7 Cross Traffic Generator

The cross-traffic generator is a C++ program that generates and sends Ethernet frames at a given rate at different priorities. For this purpose, the program models multiple virtual senders, which run as separate concurrent threads.

The individual senders are implemented using the `CrossTrafficSender` class. An instance of this class receives a `PriorityDeterminer` which sets the priority of the sent frames (see Section 6.2.2). Additionally, a target data rate, as well as the receiver's IP address and UDP port are supplied, since the sender is implemented with a UDP socket. When the sender is started, it sends Ethernet frames of size 1500 bytes to the receiver at the given data rate. For every frame, the priority is calculated by the `PriorityDeterminer`.

The individual senders are coordinated by the `CrossTrafficSenderOrchestrator` class. At initialization, this class creates multiple instances of `CrossTrafficSender` with user-provided parameters. Then, for every sender, the orchestrator creates a new thread and starts the sender inside this thread.

This way, multiple senders can operate at the same time. When the user stops the program via an interrupt signal, the orchestrator instructs all senders to save their logged data to a log file and then terminates the threads.

6.8 Logging and Feedback Channel

As evaluations should be conducted, collecting and storing various types of information is crucial. This section will first introduce the sender-receiver feedback channel used for latency and packet loss measurements. Then the structure of the general logging framework used in this work will be discussed.

6.8.1 Sender-Receiver Feedback Channel

It is of interest to measure network-related metrics, most importantly the end-to-end latency and the packet loss. Packet loss can be detected by the receiver side when sequence numbers are included in the samples, as missing sequence numbers can be translated to lost packets. However, latency measurement is more difficult. One can think of two options for how one could measure the latency with only the network available between the sender and receiver:

1. The sender includes a timestamp t_k in the k-th sample. As soon as the receiver receives the sample, it calculates
$$\ell := t_{\text{now,rcvr}} - t_k$$
to get the end-to-end latency ℓ. Here, $t_{\text{now,rcvr}}$ is the time of the receiver's real-time clock.
2. The sender includes a timestamp t_k in the k-th sample. As soon as the receiver receives the sample, it sends an acknowledgment packet containing t_k to the sender via the network. The sender then calculates
$$\ell := \frac{t_{\text{now,sndr}} - t_k}{2}$$
to get the end-to-end latency ℓ. Here, $t_{\text{now,sndr}}$ is the time of the sender's real-time clock.

Option 1's major drawback is that it requires the sender's and receiver's clocks to be perfectly synchronized to the millisecond when millisecond precision is desired. However, such a condition is difficult to achieve in a real-world system.

Option 2 has the disadvantage that it assumes that the latencies of the sample and the acknowledgment are identical. However, depending on the network load and scheduling algorithm, this might not be the case. Thus, using the network as a backchannel can lead to inaccuracies.

As we can see, calculating the end-to-end latency with only the network available is either inaccurate or only works under unrealistic conditions. A solution to these problems is extending the system model by a separate backchannel from the network. For this reason, we added a UART serial connection between the sensor-side and receiver-side Teensy 3.6 microcontrollers (c.f. Figure 5.1). The actuator logic uses this connection to send feedback to the sensor logic.

To be more specific, whenever the actuator-side microcontroller receives a new sample, it sends a message of the form

```
FB:sequenceNumber;timestamp;\n
```

to the sensor-side microcontroller via its hardware UART serial port. Here, `sequenceNumber` and `timestamp` are placeholders for the `sequenceNumber` and `currentTime` fields, which are included in the sample sent by the sensor-side Teensy 3.6 (see Section 6.1 for details).

When the sensor-side microcontroller receives such a message from the feedback channel, it calculates the end-to-end latency as follows:

$$\ell := t_{\text{now,sndr}} - t_k$$

where t_k is the `timestamp` from the feedback message and $t_{\text{now,sndr}}$ is the system time of the sensor-side microcontroller. Under the assumption that the latency of the UART connection between the microcontrollers is negligible, this ℓ is accurate. This is because both time stamps, t_k and $t_{\text{now,sndr}}$ originate from the same clock. As the UART feedback channel is implemented using Teensy's hardware serial ports [PJR23c], which support baud rates up to 4,608,000 Bd, the assumption that its latency is negligible is justified.

Additional to the latency, the sensor-side microcontroller also calculates an estimate for the packet loss: Whenever there is a gap between two received sequence numbers, this is counted as lost packets. This implies that re-ordered packets are also counted as lost. One could implement a more sophisticated loss-detection algorithm that considers packets to be lost after a certain timeout occurs, similar to TCP's retransmission algorithm [Edd22], but this would have a larger computational overhead. One can also argue that re-ordered packets do not benefit the inverted pendulum's actuator as it is only interested in the *current* angle of the pole and not in previous measurements. Thus, counting re-ordered packets as lost packets is justified.

The sensor-side microcontroller transmits the calculated latency and packet loss to the sensor-side single-board computer, which then logs this data.

6.8.2 Logging

A crucial part of this work is to develop an easy-to-use and extendable way of making evaluations of different scheduling algorithms. For this purpose, extensive logging of various statistics is vital. These statistics include values from the sensor and actuator part of the inverted pendulum, scheduling information, network statistics as well as information from the cross-traffic sender (if in use).

To make things more generalizable, we created a domain-specific logging framework in C++, which provides loggers for all named data sources. The loggers can be extended for arbitrary scheduling algorithms by using class inheritance. A class diagram of the logging framework can be seen in Figure 6.2.

The framework offers an abstract `Logger` class from which the `PendulumLogger` and `CrossTrafficLogger` classes inherit. `Logger` offers methods to save logged data to a JSON file or return a JSON string. The class uses the JSON C++ library by Niels Lohmann [Loh23] to convert the logged data into JSON.

The `PendulumLogger` class stores four lists of log entries for different sorts of logging data:

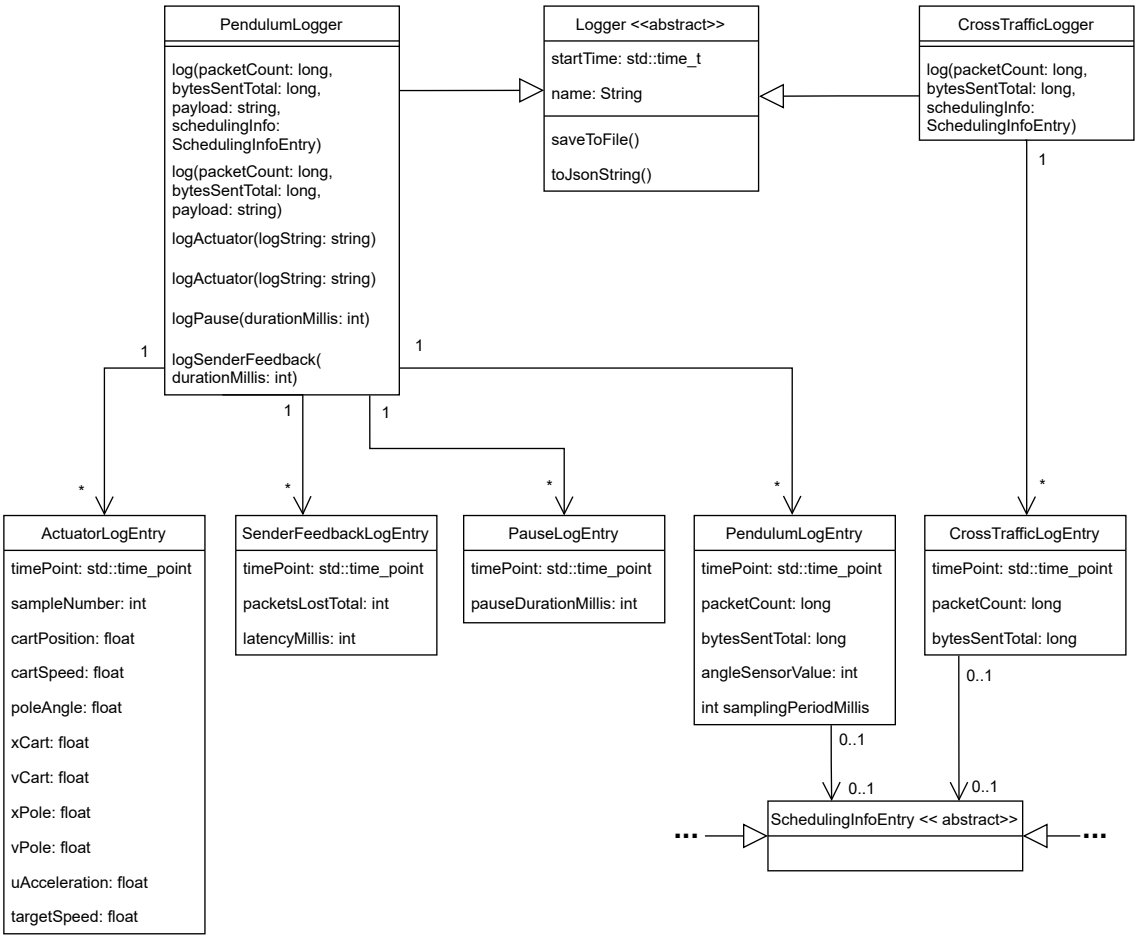


Figure 6.2: Class diagram of the logging framework

- `PendulumLogEntry`: Stores information available at the sensor logic side. This includes the angle of the pole, the sampling period, the total number of sent packets, and the total number of sent bytes since the start of the evaluation. On top of that, the entry can also store an instance of `SchedulingInfoEntry`, an abstract class that can be extended to store scheduling-specific information (see below).
- `ActuatorLogEntry`: Stores information that is generated by the actuator-side microcontroller and then sent to the actuator network component via UART after every tenth received angle value. This information includes the raw position and speed values of the cart and pole, as well as the Kalman-filtered values x_{Cart}, v_{Cart}, x_{Pole} and v_{Pole} as well as the acceleration and speed that are the output of the linear quadratic regulator.
- `PauseLogEntry`: Stores the timestamps when the motor is paused to simulate a disruption. The length of the pause in milliseconds is also stored.
- `SenderFeedbackLogEntry`: Stores information that is collected by the sender-receiver feedback channel explained in Section 6.8.1. That includes the number of lost packets and the latency with millisecond precision.

The `CrossTrafficLogger` stores a vector of `CrossTrafficLogEntry` instances. These log entries store the total number of sent packets and the total number of sent bytes. These metrics can be used to calculate the data rate and packet rate over time. Similar to the `PendulumLogEntry`, the `CrossTrafficLogEntry` also stores a `SchedulingInfoEntry` which holds scheduling-specific information.

All log entries contain time stamps so that they can be brought in relation to each other. This requires the real-time clocks of all involved single-board computers to be well-synchronized.

The `SchedulingInfoEntry` is an abstract class that can be used to create subclasses for arbitrary scheduling algorithms. This enables users to easily extend the framework to be compatible with new scheduling algorithms without modifying the rest of the code. `SchedulingInfoEntry` subtypes are supposed to store scheduling-specific information available on the sender side. For instance, the Multi-priority Token Bucket scheduling algorithm uses a token-bucket-like construction to determine the priority of the sent packets. In this case, an entry stores the current priority as well as the level of the token bucket.

7 Evaluation

In this chapter, we will evaluate the constructed framework itself, as well as the performance of scheduling algorithms using the framework. First, the types of metrics captured during the evaluations are presented. Then, some methods that allow us to compare the performance of different scheduling algorithms are introduced. Followed by that, parameters for the MPTB scheduling algorithm will be chosen. Next, evaluations investigating the correlation between sampling period and pendulum stability will be discussed. Then, the evaluations with real cross-traffic and simulated delays are explained and analyzed. Finally, we summarize our findings.

7.1 Metrics

In all evaluations, the following metrics were collected over the time of the experiment:

- Cart position on the track (m, 0 is center of track)
- Cart speed (m/s)
- Pole angle (Degrees and raw encoder value, 0° is upwards)
- Output of the Kalman filter: x_{cart} (m), v_{cart} (m/s), x_{pole} (m), v_{pole} (m/s)
- Desired acceleration and target speed as calculated by the LQR (m/s)
- Sampling period (ms)
- Number of packets and bytes sent by sensor logic and cross-traffic sender since the start of the experiment
- Total number of pendulum sample packets lost
- End-to-end latency from pendulum sensor side to actuator side (ms)
- Timestamp and duration (ms) of the intentional pauses of the motor

These metrics or combinations of them can be used to assess the performance of the scheduling algorithms under different conditions.

7.2 Comparing the Performance of Evaluations

To compare evaluation runs, reducing evaluations to a performance score is practical. As we consider scheduling algorithms to be good when they keep the pendulum stable while data usage is kept low, we decided on the following performance metrics:

- Average absolute cart position (Center position is 0 m, thus, a smaller value is better.)
- Average absolute pole angle (Upright position is 0°, thus, smaller value is better.)
- Average data rate (Smaller value is better.)
- Average cart position per average data rate: Gives how much cart stability can be achieved per byte of transferred data. (Smaller value is better.)

We also need to consider that the inverted pendulum's pole might collapse, i.e., the pole points downwards. In this case, the inverted pendulum's actuator logic stops operation as it is impossible to re-balance the pole. We refer to such a situation as a *crash* in this work. Of course, such crashes are strongly unwanted behavior. To depict this in the performance metrics, we introduce a *punishment*: Before calculating the average, we extend the measurement series by a worst-case value ξ. More formally, let the normal average without punishment be $\mathrm{Avg}_{\mathrm{unpunished}}$, the experiment starts at time $t_0 = 0$, the crash happens at t_{crash} and the intended end of the experiment (if no crash happens) is t_{end}. Then it is

$$\mathrm{Avg}_{\mathrm{punished}} = \frac{t_{\mathrm{crash}}}{t_{\mathrm{end}}} \cdot \mathrm{Avg}_{\mathrm{unpunished}} + \frac{t_{\mathrm{end}} - t_{\mathrm{crash}}}{t_{\mathrm{end}}} \cdot \xi$$

the punished average value.

ξ depends on the metric that the average is calculated on. For cart position and pole angle, we decide on the following values

$$\xi_{\mathrm{cartPosition}} = 0.6\ \mathrm{m}, \qquad \xi_{\mathrm{poleAngle}} = 180^\circ$$

$\xi_{\mathrm{cartPosition}}$ is chosen this way because when the cart hits the track's left or right limit switch, it is at position ±0.6 m. $\xi_{\mathrm{poleAngle}} = 180^\circ$ is the angle of the pole pointing downwards, which is a realistic angle after a crash, since the pole can rotate 360° around the rotary encoder's axis.

7.3 Parameters for MPTB

An MPTB scheduling with eight priority queues is considered, as this equals the number of priorities representable by the IEEE 802.1Q VLAN tag. Further, we assume a 1:1 mapping between severity levels S_j and priority classes C_i, i.e., the number of severity levels equals the number of priority classes, and it is $Class(S_0) = C_7, \ldots, Class(S_7) = C_0$.

For MPTB, the following parameters need to be chosen:

- The cost to send one byte $Cost(C_i)$ for each priority class C_i
- The thresholds $Th(S_j)$ for each severity level S_j
- The sender's contract $(C_{\mathrm{init}}, b, r)$.

We choose these parameters with the help of formulas given in Section 5.3.5. Thus, only the sender's contract (C_{init}, b, r) and per-priority data rates $r_0, \ldots, r_7$ must be chosen. We choose the per-severity thresholds $b_0, \ldots, b_7$ dependent on b as explained in Section 5.3.5.

As the pendulum is a time-critical system, we set $C_{\text{init}} := C_7$, i.e., initially, the pendulum's data stream has the highest priority.

While one could choose r and $r_0, \ldots, r_7$ directly, it is more intuitive to choose sampling periods instead. The sampling period T_i determines the time between two sample transmissions. As the samples have a constant frame size of 78 bytes, one can therefore calculate the data rate from the sampling period using the following formula:

$$r_i = \frac{78 \text{ bytes}}{\frac{T_i}{1000}}$$

where T_i is in milliseconds and r_i is in bytes per second. It is worth mentioning that it must be $r_7 = r$ since the sustainable data rate r of the stream must be equal to the data rate r_7 that guarantees the stream to stay in its initial priority class C_7.

Hence, in total, we only need to choose the per-priority sampling periods $T_0, \ldots, T_7$ and the contract's bucket size b. All cost and threshold parameters of MPTB can be calculated from these nine parameters. For the evaluation, we chose three different sets of per-priority sampling periods, which we call P_{strict}, P_{medium}, and P_{generous}. The chosen per-period sampling periods T_i and their corresponding per-priority data rates r_i are listed in Table 7.1. P_{strict} only allows low data rates in higher priority classes, i.e., the stream quickly gets degraded to a lower priority. In contrast, P_{generous} allows high data rates in high priorities. P_{medium} is, as its name implies, in between strict and generous priority.

Note that the inverted pendulum has a minimum sampling period of ten milliseconds, corresponding to the highest data rate. Thus, in P_{medium} and P_{generous}, the inverted pendulum's stream cannot drop into a priority class lower than C_3 and C_5, respectively, as all lower priority classes are associated with sampling periods below 10 ms.

In addition to the per-priority sampling periods, we also choose three different token bucket sizes b, which we call B_{strict}, B_{medium} and B_{generous}. We choose the bucket sizes in numbers of samples, which is possible since the frame size of one sample is constant at 78 bytes. The concrete values for these parameters (in samples and bytes) are shown in Table 7.2.

In total, there are $3 \times 3 = 9$ combinations of the different per-priority data rates and bucket sizes. Using the formulas from Section 5.3.5, the selected parameters can be converted into costs and thresholds for the MPTB scheduling algorithm. The results of these conversions can be seen in Table 7.3 and Table 7.4.

7.4 Correlation Between Sampling Period and Pendulum Stability

It is of interest whether a lower sampling period correlates with a higher stability of the inverted pendulum. To assess this hypothesis, we conducted measurements with the following sampling periods: 10 ms, 30 ms, 50 ms, 70 ms, 90 ms, 100 ms, 110 ms, 130 ms, and 150 ms. In every evaluation, the pendulum was active for 120 seconds while the pendulum's motor stopped every 20

	Sampling Period (ms)		
	P_{strict}	P_{medium}	P_{generous}
T_7	90	75	50
T_6	80	59	30
T_5	70	43	10
T_4	60	26	9
T_3	50	10	8
T_2	40	9	7
T_1	30	8	6
T_0	20	7	5
T	90	75	50

(a) Per-Priority sampling periods

	Data Rate (bytes/s)		
	P_{strict}	P_{medium}	P_{generous}
r_7	866.7	1040.0	1560.0
r_6	975.0	1322.0	2600.0
r_5	1114.3	1814.0	7800.0
r_4	1300.0	3000.0	8666.7
r_3	1560.0	7800.0	9750.0
r_2	1950.0	8666.7	11142.9
r_1	2600.0	9750.0	13000.0
r_0	3900.0	11142.9	15600.0
r	866.7	1040.0	1560.0

(b) Per-Priority Data Rates

Table 7.1: Per-priority sampling periods and their corresponding data rates as a basis for the calculation of the cost and threshold parameters for MPTB

	B_{strict}	B_{medium}	B_{generous}
b (samples)	75	300	500

(a)

	B_{strict}	B_{medium}	B_{generous}
b (bytes)	5850	23400	39000

(b)

Table 7.2: Bucket sizes in samples and bytes (size of one sample: 78 bytes)

	Cost		
Priority	P_{strict}	P_{medium}	P_{generous}
C_7	1.0	1.0	1.0
C_6	0.89	0.79	0.6
C_5	0.78	0.57	0.2
C_4	0.67	0.35	0.18
C_3	0.56	0.13	0.16
C_2	0.44	0.12	0.14
C_1	0.33	0.11	0.12
C_0	0.22	0.09	0.1

Table 7.3: Cost parameters for MPTB. These values were calculated from the per-priority data rates shown in Table 7.1.

Severity	Threshold P_{strict}	P_{medium}	P_{generous}
S_0	0	0	0
S_1	-66.67	-59.0	-45.0
S_2	-125.0	-102.0	-60.0
S_3	-175.01	-128.0	-73.5
S_4	-216.67	-138.0	-85.5
S_5	-250.01	-147.0	-96.0
S_6	-275.01	-155.0	-105.0
S_7	$-\infty$	$-\infty$	$-\infty$

(a) Strict bucket B_{strict}

Severity	Threshold P_{strict}	P_{medium}	P_{generous}
S_0	0	0	0
S_1	-266.68	-236.01	-180.0
S_2	-500.02	-408.0	-240.0
S_3	-700.02	-512.0	-294.0
S_4	-866.7	-552.0	-342.0
S_5	-1000.04	-588.0	-384.0
S_6	-1100.04	-620.0	-420.0
S_7	$-\infty$	$-\infty$	$-\infty$

(b) Medium bucket B_{medium}

Severity	Threshold P_{strict}	P_{medium}	P_{generous}
S_0	0	0	0
S_1	-444.46	-393.34	-300.0
S_2	-833.36	-680.0	-400.0
S_3	-1166.71	-853.34	-490.0
S_4	-1444.5	-920.0	-570.0
S_5	-1666.73	-980.0	-640.0
S_6	-1833.4	-1033.34	-700.0
S_7	$-\infty$	$-\infty$	$-\infty$

(c) Generous bucket B_{generous}

Table 7.4: MPTB threshold values for all nine combinations of bucket sizes and per-priority data rates. These value were calculated from the per-priority data rates in Table 7.1 and the bucket sizes in Table 7.2.

seconds for 800 ms to create artificial disruptions. The network introduced in Section 4.1 is used to transmit the samples. However, neither cross-traffic nor artificial latencies are used. Thus, the network does not introduce significant negative influences on the quality of service.

Figure 7.1 shows the cart position and pole angle over the time of the evaluations for all evaluated sampling periods. It can be seen that for sampling periods of 30 ms, 50 ms, and 70 ms, the pendulum stays relatively stable, even at disturbances (marked by the grey bars). For sampling periods at and above 90 ms, it can clearly be seen that the pendulum behaves less stable and even crashes before the end of the experiment for sampling periods of 130 and 150 milliseconds. For the sampling period of 150 milliseconds it can be seen that the pendulum is extremely unstable and crashes even before the first disruption.

One interesting aspect is the behavior of the 10 ms sampling period evaluation. Here we can see that the pendulum's cart position and angle start oscillating with increasing amplitude. We repeated this experiment for 10 ms several times and always got the same result: After 90 to 100 seconds, the pendulum starts oscillating and in most cases also crashes. We were unable to determine the exact reason for this behavior; however, we assume that a too-high sampling rate results in the overcorrection of the pendulum's state. Another possible explanation is that changing the pendulum's acceleration every ten milliseconds correlates with the natural frequency of some of the inverted pendulum's materials. This causes the materials to oscillate, which introduces instability.

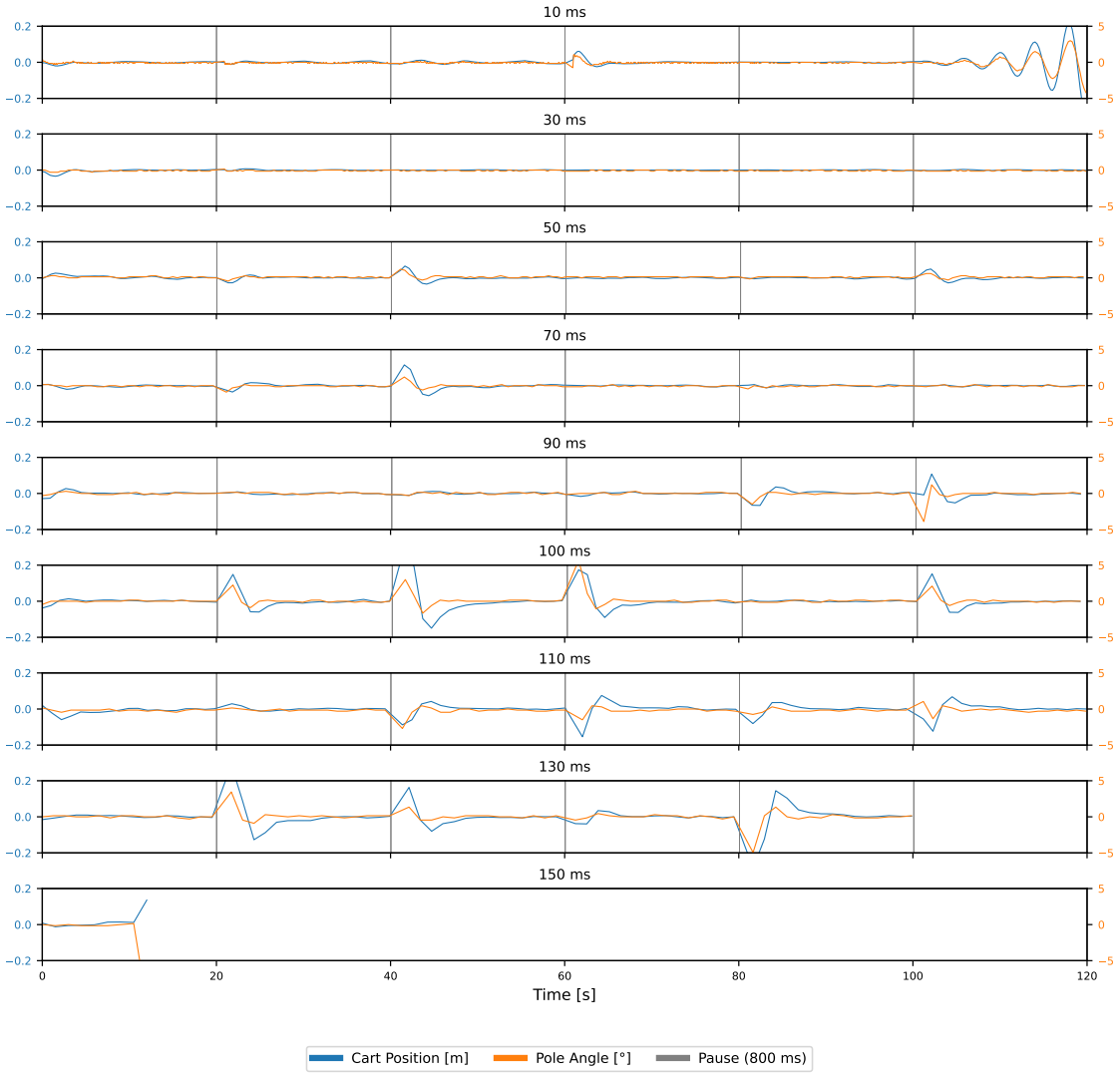


Figure 7.1: Evaluations at different constant sampling rates without network influence

It can also be seen that the pole angle and cart position have a strong positive correlation. Therefore, we will primarily consider the cart position in the rest of this work.

To better understand the influence of the sampling period, Figure 7.2 shows the average absolute cart position (with a punishment of $\xi_{\text{cartPosition}}$ after crashes). As can be seen, the stability of the pendulum monotonically increases with increasing sampling periods above 10 ms. The sampling period of 30 ms yields the best performance, while 50 and 70 ms also perform well. Sampling periods of 130 and 150 ms lead to crashes. Thus, we can conclude a positive correlation between decreasing sampling period and increasing stability of the inverted pendulum. However, we can also see that this correlation is not very strong, especially in the range of 50 to 90 milliseconds. Very low sampling periods (10 ms) do not correlate with higher stability but instead lead to oscillations.

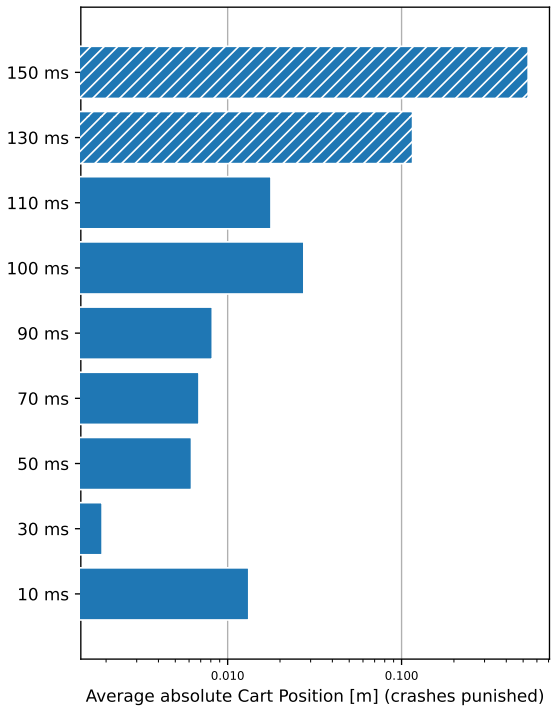


Figure 7.2: Average absolute cart position of the evaluations at different constant sampling rates without network influence. Crashes were punished with $\xi_{\text{cartPosition}}$. Crashed runs are marked with hatched bars. Logarithmic scale.

7.5 Evaluation With Real Cross Traffic

We must examine whether evaluations with real cross-traffic can lead to relevant results with the given setup. For this purpose, we conduct some measurements to assess how the cross traffic and queue size influence latency and packet loss.

For the experiment, we use the hardware system introduced in Section 5.1. The cross-traffic generator sends a total of 11 Mbit/s divided equally across the eight priority classes, i.e., on each priority, there is a stream sending 1.375 Mbit/s. One might ask why the cross traffic generator sends 11 Mbit/s when the critical bottleneck link of the system can only handle 10 Mbit/s. However, experiments were also conducted with 10 Mbit/s, leading to less reliable latency behavior of the system.

The switch runs a strict-priority scheduling as used by MPTB. The queue size is a parameter that influences the latency of the individual priority classes. This parameter was chosen as eight frames per queue, i.e., the switch can buffer eight frames per priority class. In theory, this should lead to a maximum per-priority latency of

$$\delta_k = (8-k) \cdot \frac{8 \text{ frames} \cdot 1500 \text{ Byte/frame} \cdot 8 \text{ Bit/Byte}}{10,000,000 \text{ Mbit/s}} = 9.6(8-k) \text{ ms}$$

for priority C_k, i.e. $\delta_7 = 9.6$ ms, $\delta_6 = 19.2$ ms, $\delta_5 = 28.8$ ms, $\delta_4 = 38.4$ ms, $\delta_3 = 48.0$ ms, $\delta_2 = 57.6$ ms, $\delta_1 = 67.2$ ms and $\delta_0 = 76.8$ ms.

The pendulum sensor logic transmits a sample every 50 milliseconds during the experiment. This sample is received by the pendulum actuator logic and then forwarded to the sensor logic via the UART feedback channel described in Section 6.8.1. With this method, the system's end-to-end latency and packet loss can be measured precisely as we do *not* use the network as a backchannel. Note that the physical inverted pendulum is *not* active during the experiment, as we are not interested in the pendulum's performance but only in the system's latency and packet loss behavior.

An evaluation was conducted for each priority class C_k in which the pendulum sensor logic transmits samples with priority C_k for 60 seconds. For comparison, an additional evaluation was performed in which the cross-traffic generator is turned off, and the priority class is C_7.

The results regarding latency are shown as box plots in Figure 7.3. As can be seen, the actual latency behavior deviates significantly from the theoretical latency bounds calculated above: For priorities 7 to 3, there is only a weak correlation between decreasing priority and increasing latency. Only for priorities 3 to 0, a clear increase in latency can be seen when the priority decreases. Also, the median latencies are significantly higher than calculated: For C_7, the median latency is 17 ms instead of 9.6 ms. For C_0, the median is even 37 ms higher than intended, with 114 ms instead of 76.8 ms. Partially, this can be explained by an additional latency introduced by the network interface's ring buffer. This ring buffer was set to size 80 packets, the smallest size possible. Assuming a frame size of 1500 bytes, this can introduce an additional delay of up to 96 ms. However, we can assume that the ring buffer is only partially used when looking at the resulting latencies.

Additionally, we can observe that the latency fluctuates heavily within one priority class. For instance, in C_6, the latency has a standard deviation of $\sigma_6 = 9.2$, and for C_1, the standard deviation is even $\sigma_1 = 29.1$. These fluctuations likely come from the fact that the priority queues are not always filled completely.

Figure 7.4 shows the packet losses during the experiments. As can be seen, there is no significant packet loss for priorities 7 to 3. However, for priority classes 2, 1, and 0, there is significant packet loss and also a strong correlation between increasing packet loss and decreasing priority.

Due to the weak correlation between priority class and latency for high priorities and due to the high fluctuation in latency, we decided that it is not helpful to conduct further evaluations with real cross traffic. MPTB is based on the assumption that a higher priority class guarantees lower latency, but this cannot be modeled sufficiently well by this system. We have also tried other parameters for the queue sizes and cross-traffic generation, but these did not yield more promising results. Another problem is that even at low priorities, the latency is relatively high for the application of an inverted pendulum. As seen in Section 7.6, the inverted pendulum becomes highly unstable for latencies above 30 milliseconds. However, as the box plots in Figure 7.3 show, all priority classes except C_7 and C_6 have median latencies above 30 ms.

7.6 Evaluation With Simulated Delays

As the evaluations in the previous section show, with real cross-traffic, there is only limited control over the per-hop latency at the switch. Therefore, to have more fine-grained control over these latencies, we omit the cross traffic and instead apply a deterministic artificial delay to the output of each queue as described in Section 6.6.3. The system described in Chapter 4 is used for the

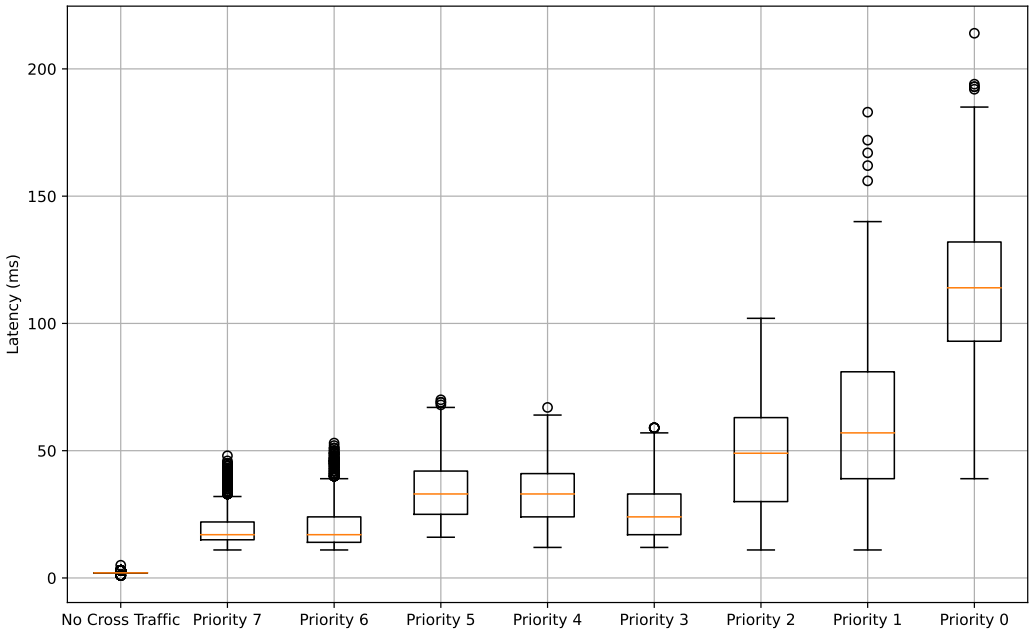


Figure 7.3: Box plots showing the end-to-end latencies of the pendulum sender transmitting in different priority classes. During the experiment, cross traffic with 11 Mbit/s equally distributed across the eight priorities was active (except for the leftmost box plot). Queue length per priority: 8 packets. Experiment duration: 60 seconds per priority. The box shows the interquartile range (IQR) and the whiskers show 1.5 · IQR. Outliers are marked by dots.

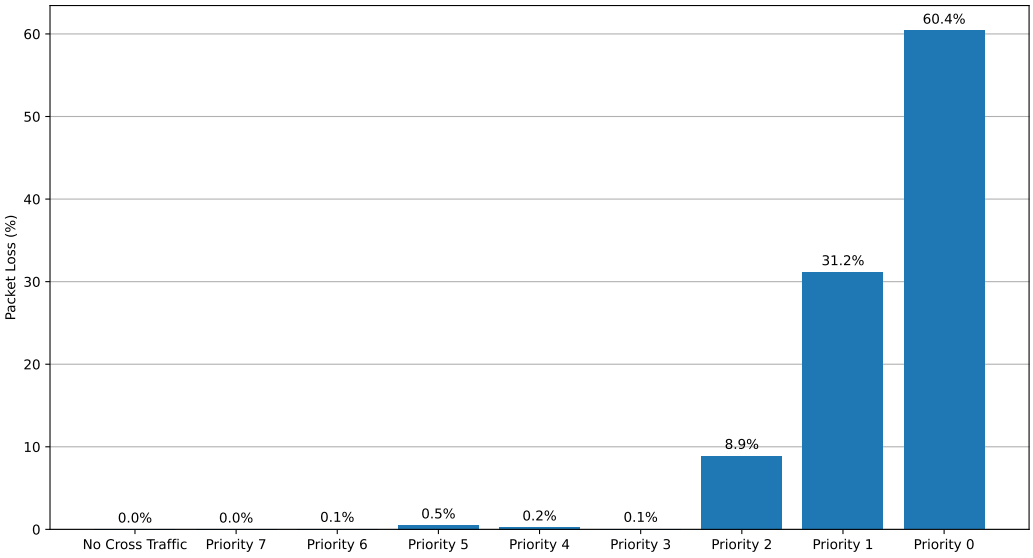


Figure 7.4: Packet losses of the pendulum sender while transmitting in different priority classes. During the experiment, cross traffic with 11 Mbit/s equally distributed across the eight priorities was active (except for the leftmost bar). Queue length per priority: 8 packets. Experiment duration: 60 seconds per priority.

evaluations. However, the cross-traffic generator and receiver components are not involved. In the following, we will first explain our evaluation methods and then present and interpret the evaluation results.

7.6.1 Evaluation Methods

This section will explain the evaluation methods for the FIFO and MPTB scheduling algorithms.

FIFO

To simulate a FIFO scheduling without real cross-traffic, a constant delay is applied to all frames that are processed by the switch. Two evaluations were performed, one with a dynamic and one with a static sampling rate:

- **FIFO DSR:** Dynamic sampling rate with the following network latencies:
  5 ms, 10 ms, 15 ms, 20 ms, 25 ms, 30 ms, 35 ms, 40 ms
- **FIFO CSR:** Constant sampling rate with all 24 combinations of the following sampling periods and network latencies:
  - Sampling periods: 10 ms, 30 ms, 50 ms, 70 ms, 90 ms, 100 ms
  - Network Latencies: 5 ms, 15 ms, 25 ms, 35 ms

Note that fewer latency values were used for the evaluation with a constant sampling rate than for DSR. Otherwise, $8 \times 6 = 48$ evaluations would have had to be carried out, which was too time-consuming. For every configuration, the pendulum was left balancing for 120 seconds while an artificial pause of 800 ms was inserted every 20 seconds.

MPTB

MPTB has multiple priority queues, each with a different maximum latency guarantee. To simulate this without real cross traffic, a fixed per-queue delay is applied. We decided on three sets of delays which we call L_{low}, L_{medium}, and L_{high}. The per-queue delays are listed in Table 7.5.

For each set of delays, for every set of per-priority data rates in Table 7.1 and for all bucket sizes in Table 7.2 an evaluation of the pendulum's performance was conducted. In total, this yields $3 \times 3 \times 3 = 27$ different configurations.

For every configuration, the pendulum was left balancing for 120 seconds while an artificial pause of 800 ms was inserted every 20 seconds. All evaluations used the adaptive sampling rate determination described in Section 5.5. An evaluation of MPTB with a constant sampling rate was omitted since the dynamic priority adaption of MPTB cannot yield any benefit at constant data rates, as the priority would be constant too in such cases.

Priority	Delay (ms)		
	L_{low}	L_{medium}	L_{high}
C_7	2	5	10
C_6	4	10	20
C_5	6	15	30
C_4	8	20	40
C_3	10	25	50
C_2	12	30	60
C_1	14	35	70
C_0	16	40	80

Table 7.5: Simulated delays per priority class of the MPTB scheduling algorithm

7.6.2 Evaluation Results

FIFO CSR Figures 7.5, 7.6, 7.7, 7.8, 7.9, and 7.10 show the evaluations of FIFO scheduling with 10, 30, 50, 70, 90 and 100 milliseconds sampling period respectively. Each figure shows the cart position over time for four simulated queueing delays: 5, 15, 25, and 35 ms.

For a sampling period of 10 milliseconds (see Figure 7.5), the phenomenon discussed in Section 7.4 can be observed too: Despite the low sampling period, the pendulum starts oscillating and eventually crashes in all four evaluations. A correlation between latency and stability cannot be observed here.

For all other evaluations, it can be seen that there is at least some correlation between latency and stability of the cart: While for 5 ms and 15 ms, the pendulum is mostly stable, crashes and strong oscillation of the cart are common for a latency of 25 and 35 milliseconds. For a latency of 35 milliseconds, a crash even happens in every experiment.

In Figure 7.15, the average absolute cart position of all evaluations is shown as a bar chart. Here we can see better that the correlation between latency and cart stability is only weak since for 5, 15, and 25 ms, many outliers oppose the hypothesis of a strong correlation.

However, when looking at the average absolute pole angles depicted as a bar graph in Figure 7.16, one can see a correlation between the stability of the pole and the latency: For all evaluations with constant latency and constant sampling period except for 10 and 30 milliseconds sampling period, we see that the average absolute angle increases monotonically with increasing latency.

It can also be observed that the likelihood of crashes is increased at higher sampling periods: While for sampling periods of 30 and 50 milliseconds, a crash only happens for a latency of 35 milliseconds, we can see that for sampling periods 70, 90, and 100 ms there is a crash at 25 milliseconds as well.

Interestingly, the best-performing sampling period is not the lowest (10 ms) or the second-lowest (30 ms), but it is 50 milliseconds: For both the cart position and the pole angle, 50 milliseconds gives the best results.

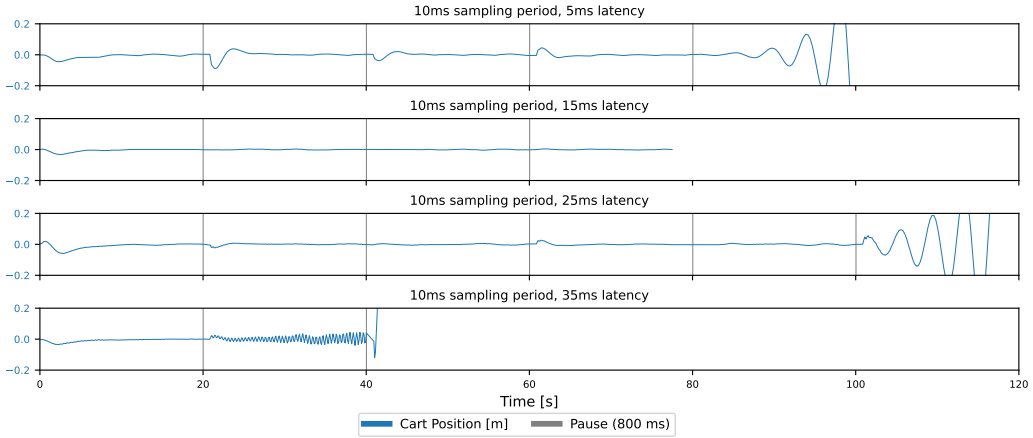


Figure 7.5: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 10 ms

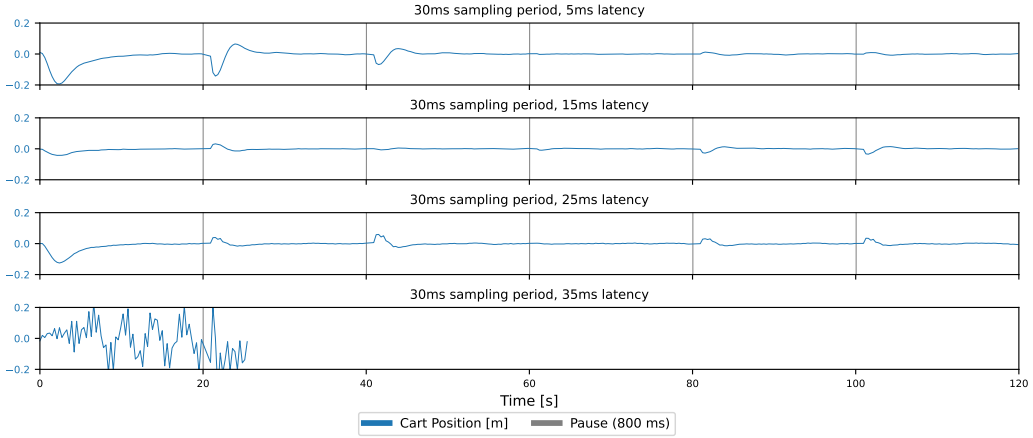


Figure 7.6: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 30 ms

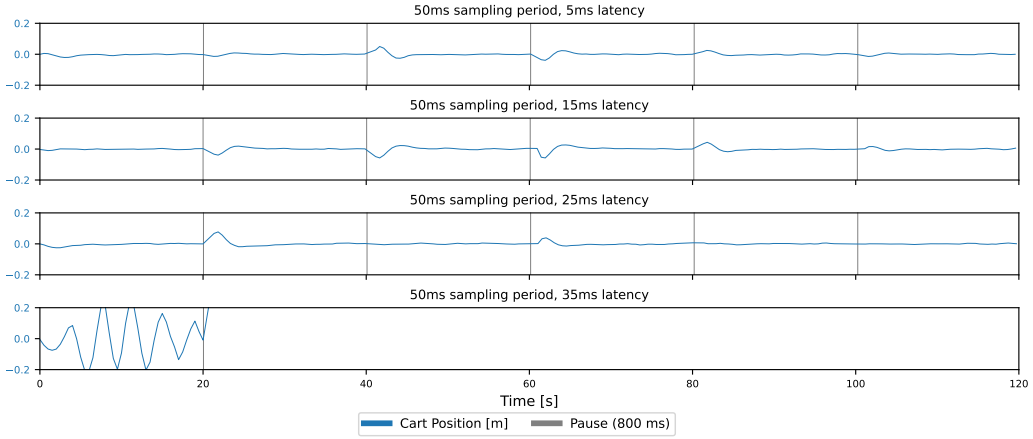


Figure 7.7: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 50 ms

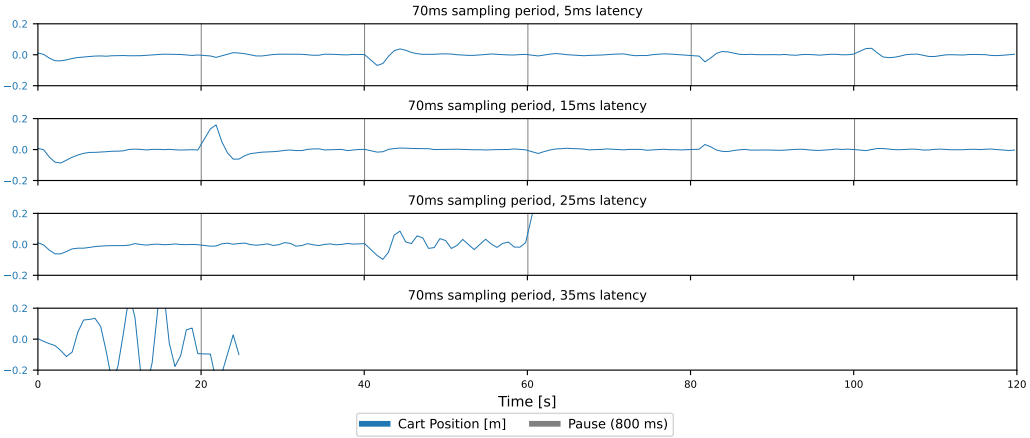


Figure 7.8: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 70 ms

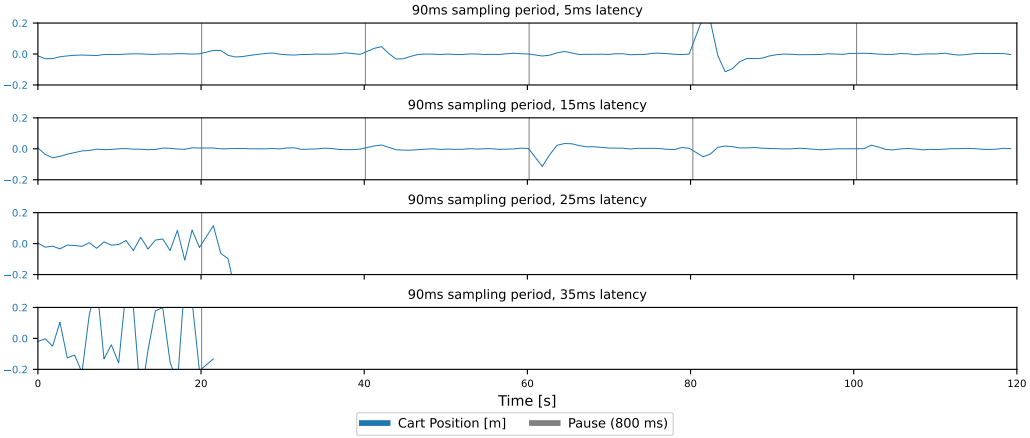


Figure 7.9: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 90 ms

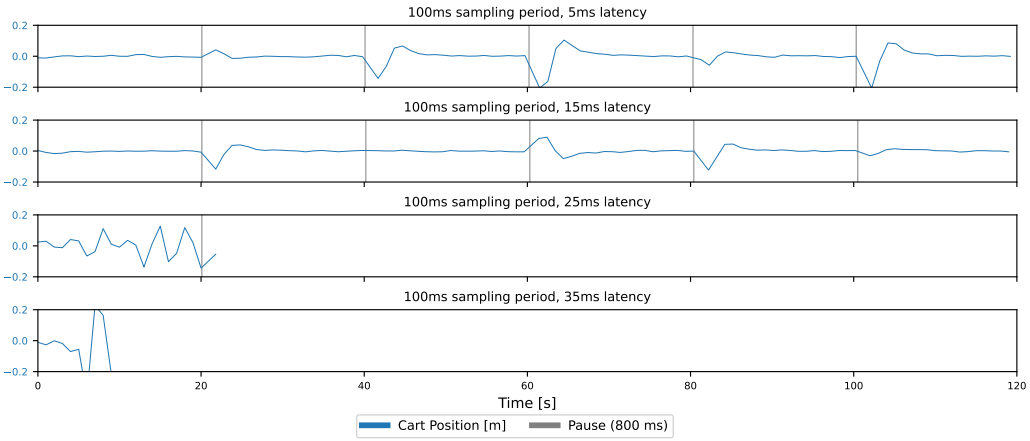


Figure 7.10: Cart position during FIFO scheduling evaluations with different simulated delays at a sampling period of 100 ms

FIFO DSR Figure 7.11 shows the cart position and data rate of the evaluations with simulated FIFO scheduling and dynamic sampling rate adaption. The FIFO scheduling was simulated with latencies of 5, 10, 15, 20, 25, 30, 35, and 40 milliseconds.

As expected with the dynamic sampling rate adaption, the data rate increases after almost every artificial disruption of the pendulum. The reason is that the pole angle changes significantly after the disruption, which causes the sender to increase the sampling rate. As more samples are transmitted, the data rate increases as well, of course.

One can also see that with increasing latency, the stability of the pendulum decreases, and the average data rate increases: For latencies 5, 10, 15, and 20 milliseconds the pendulum does not crash, and the data rate only increases briefly after the artificial disruptions. For 25 and 30 milliseconds of latency, the pendulum crashes after the third disruption, and for 35 and 40 milliseconds, even after the first. The data rate is permanently increased for 30, 35, and 40 milliseconds of latency as the pole angle is permanently unstable.

To compare FIFO with dynamic sampling rate to FIFO with constant sampling rate, we can look at the performance bar charts in Figure 7.15 and Figure 7.16, showing the average absolute cart position and pole angle, respectively. For better overview, there is also a diagram containing the performance values for only the most relevant evaluations in Figure 7.14. For a latency of 5 ms, we see that the cart position stability of FIFO DSR is comparable to that of FIFO CSR with a sampling period of 70 ms. The pole angle stability is similar to that of FIFO CSR with a 90 ms sampling period. With 15 ms of latency, the average cart position of FIFO DSR is higher than that of FIFO CSR for all evaluated sampling periods. The pole angle performance is comparable to that of FIFO CSR with a sampling period of 70 milliseconds. At latency 25 ms, the (crashed and thus punished) cart position performance of FIFO DSR is almost identical to that of FIFO CSR with a 70 milliseconds sampling period. The same holds for the pole angle performance.

In summary, FIFO with dynamic sampling rate adaption performs similarly to FIFO with a constant sampling rate of 70 milliseconds. However, when looking at the average data rates shown in Figure 7.17, we can see that FIFO CSR 70 ms has slightly lower data rates than FIFO DSR. Thus, we can conclude that FIFO with dynamic sampling rate adaption yields no significant benefit over FIFO CSR.

MPTB Figures 7.12 and 7.13 show the cart position, priority, and data rate of the evaluations with the MPTB scheduling algorithm in combination with dynamic sampling rate adaption for the L_{medium} and L_{high} latency sets. The values of the latency sets are shown in Table 7.5. Each figure shows nine evaluations — one for each combination of the per-priority data rates and bucket sizes listed in Table 7.1 and Table 7.2. To avoid confusion, the priority is given as described in Chapter 5: 7 is the highest and 0 is the lowest priority. In the real implementation, it is 0 the highest and 7 the lowest priority, as this is the convention within the Linux kernel. The same evaluation was also conducted for the L_{low} latency set. However, we exclude it since it did not give any significant results, probably because the differences in latency are too small. Nonetheless, interested readers can find the results in Figure A.1.

Both figures show that from strict to medium to generous bucket sizes (B) and per-priority data rates (P), the amount of time the sender spends in lower priorities decreases.

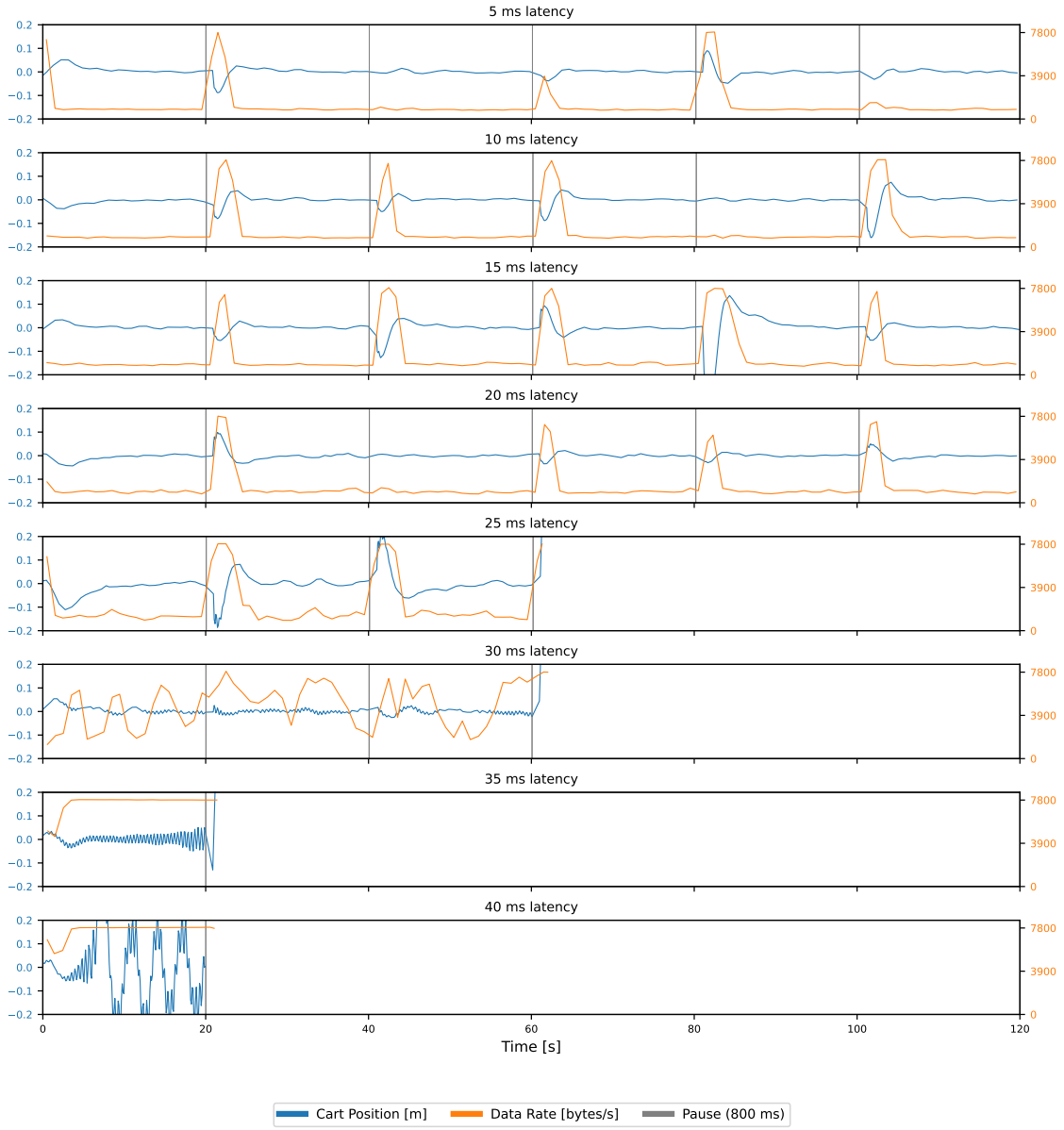


Figure 7.11: FIFO scheduling evaluations with different simulated latencies with dynamic sampling rate adaption

With generous bucket size (B_{generous}), it can be observed that the priority never drops for L_{medium} and L_{high}. With a medium bucket (B_{medium}), the priority only drops in the evaluations with strict and medium per-priority data rates. A strict bucket causes the priority to drop by at least two priority classes in every evaluation. This shows that increases in the data rate of the pendulum are only temporary and can therefore be covered up by large enough buckets.

It can also be seen that for P_{strict}, the priority permanently decreases in all evaluations except the evaluation with the configuration (L_{high}, B_{generous}, P_{strict}). A permanent decrease in priority implies that the contract is permanently violated due to above-contract data rates.

One can observe that a decrease in priority can lead to instability of the cart. This again increases the data rate due to dynamic sampling rate adaption, which again causes a drop in priority. This behavior eventually leads to a crash, as can be seen with the configurations (L_{medium}, B_{strict}, P_{strict}), (L_{high}, B_{strict}, P_{strict}) and (L_{high}, B_{medium}, P_{medium}). In the configuration (L_{medium}, B_{strict}, P_{strict}) (see Figure 7.12), this phenomenon can be observed best: After the third disruption, the priority decreases from 6 to 3. Followed by this, the cart starts oscillating lightly, but the priority increases to 4 again nonetheless. Then, after the fourth disruption, the priority drops from 4 to 2. About seven seconds after the disruption, the cart starts oscillating with a strongly increasing amplitude. This causes the priority to drop to C_0 (best effort), what causes the cart to oscillate even more. Then, after the next disruption, the pendulum finally crashes.

This shows that too strict choices of the parameters for MPTB can make it counter-productive in the scope of a networked control system: When the NCS reaches a critical state causing an increase in data rate, MPTB punishes the NCS by degrading its priority. This implies a higher latency which can put even more stress on the NCS, causing it to reach a more critical state. This negative self-amplifying effect needs to be kept in mind when using MPTB.

The configurations (L_{medium}, B_{strict}, P_{medium}) and (L_{high}, B_{strict}, P_{generous}) use MPTB in its intended way: In both evaluations, it can be observed that the priority temporarily decreases after every disruption but increases again after the inverted pendulum has recovered from the disruption. Thus, the pendulum shortly violates its contract in critical states but complies with it again after reaching a more stable state.

When looking at the performance bar graphs in Figure 7.15 and Figure 7.16, we can see that for L_{high} there is a clear impact of per-priority data rates and bucket sizes on both, the cart position and the pole angle stability: With increasing bucket size and per-priority data rates the pendulum becomes a lot more stable. For L_{medium}, there is a tendency towards the same behavior as for L_{high}. However, for the low-latency set L_{low}, there seems to be no clear impact of per-priority data rates and bucket sizes. The per-priority latencies in L_{low} are probably too low to impact the performance of the pendulum significantly. Thus, a higher priority does not result in better performance in this case.

To assess whether MPTB is beneficial, comparing it to the FIFO scheduling discussed before is helpful. As the configurations (L_{medium}, B_{strict}, P_{medium}) and (L_{high}, B_{strict}, P_{generous}) use MPTB like intended (reduce priority only temporary at disruptions), we will consider them as representatives of MPTB.

(L_{medium}, B_{strict}, P_{medium}) performs better than FIFO DSR 15 ms latency both in terms of cart position and in terms of pole angle. This speaks in favor of MPTB since, when comparing the average data rates, MPTB has an almost identical average data rate as said FIFO DSR evaluation.

The same MPTB configuration also has a better pole angle performance than FIFO CSR at 5 ms latency, 30 ms sampling period despite MPTB's average data rate being significantly lower. This implies that MPTB in combination with an adaptive sampling rate can perform better than FIFO scheduling at lower average data rates and partially higher latencies.

For the MPTB configuration (L_{high}, B_{strict}, P_{generous}), we can see that its performance is almost the same as that of FIFO DSR 15 ms latency, and also the average data rates are almost identical. Said MPTB configuration spends time only in priorities 5, 6, and 7, which correspond to latencies of 30, 20, and 10 milliseconds; thus, a similar average latency is achieved.

Besides these observations, we can see that, in general, that the evaluations with MPTB has significantly fewer crashes than the evaluations with FIFO scheduling: Four of eight evaluations of FIFO DSR and twelve of 24 evaluations of FIFO CSR crash before the end of the experiment. Meanwhile, only three of the 27 configurations of MPTB lead to a crash. Part of the reason is obviously that most MPTB configurations spend large shares of their time in higher priorities which have low latencies. However, there are also evaluations of MPTB where the sender spends significant parts of the time in lower priorities and still does not crash. For example, the configuration (L_{high}, B_{strict}, P_{generous}) spends time in priorities 5 and 6, corresponding to latencies of 30 and 20 milliseconds. Looking at the FIFO CSR and DSR evaluations, we can see that a latency above 20 ms is a significant risk for crashing. Therefore, we can conclude that MPTB is less likely to lead to crashes than FIFO as the sender only faces high latencies for part of the time and not constantly.

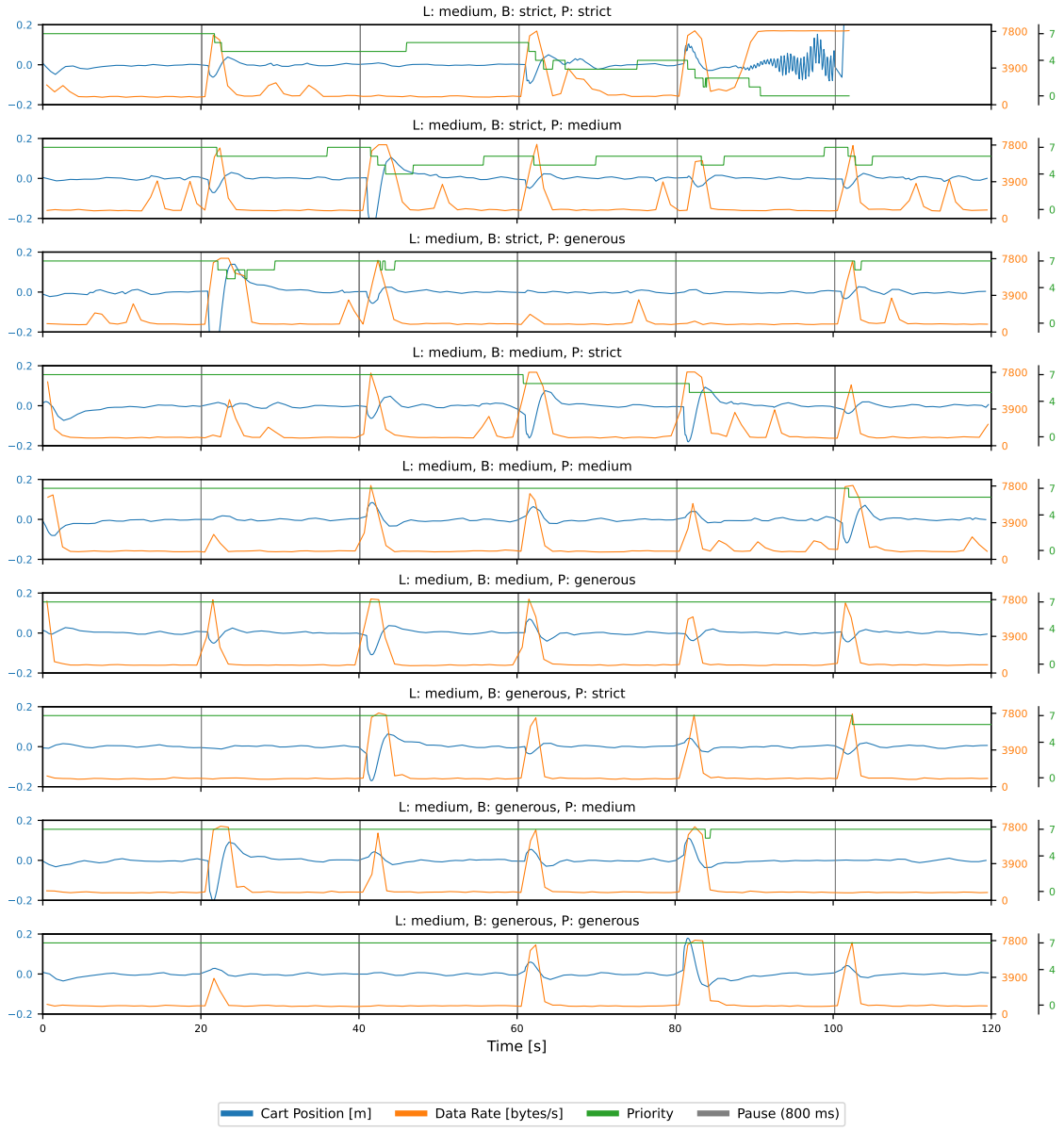


Figure 7.12: Multi-priority Token Bucket evaluations with the L_{medium} set of latencies delays (see Table 7.5). All nine combinations of bucket sizes (B) (Table 7.2) and per-priority data rates (P) (Table 7.1) were evaluated. Dynamic sampling period adaption was used.

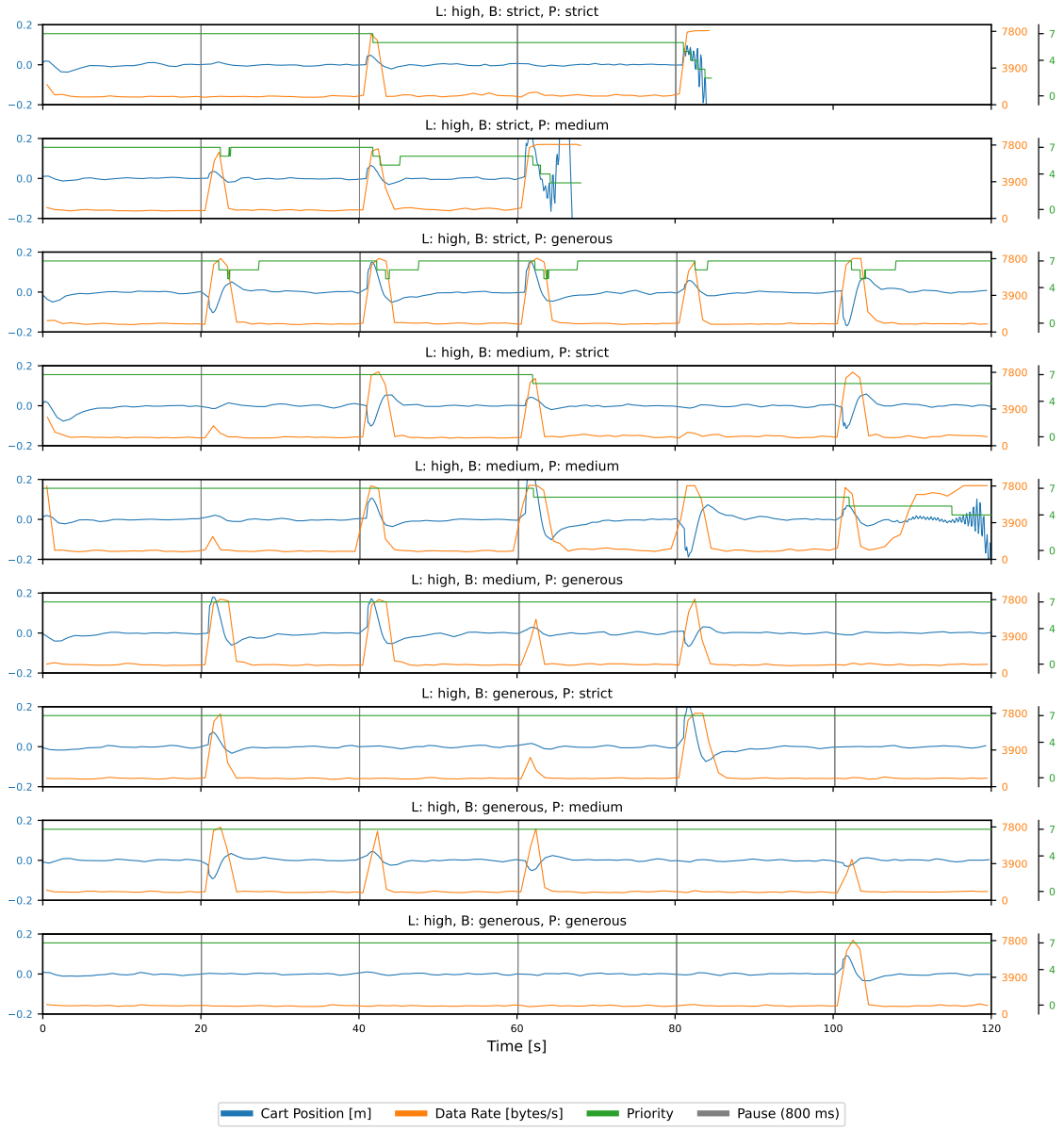
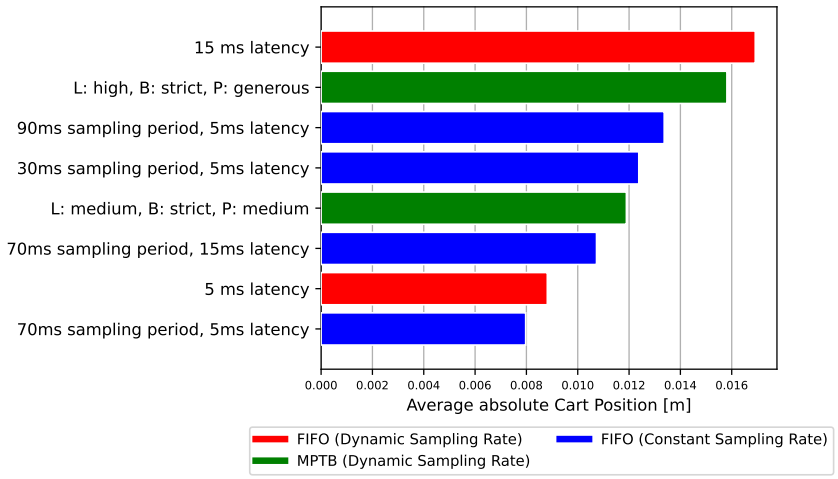
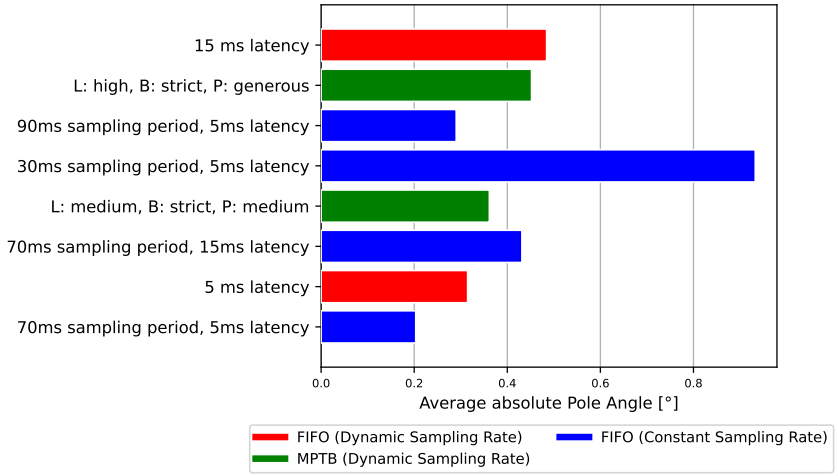


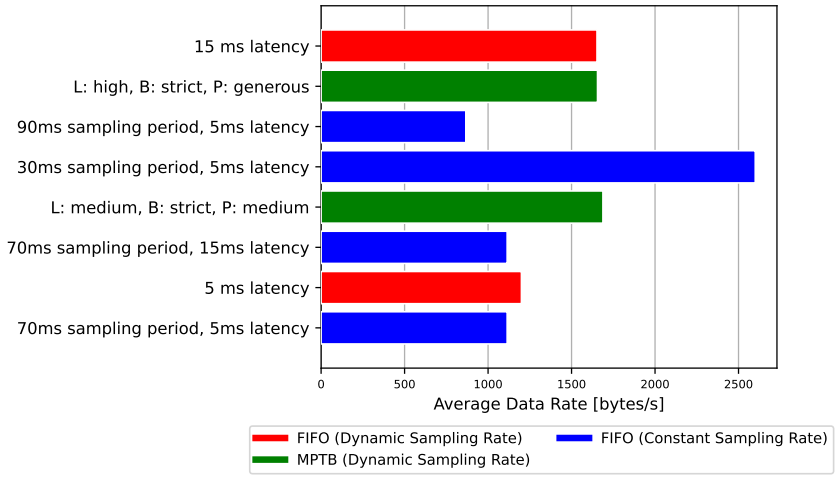
Figure 7.13: Multi-priority Token Bucket evaluations with the L_{high} set of simulated latencies (see Table 7.5). All nine combinations of bucket sizes (B) (Table 7.2) and per-priority data rates (P) (Table 7.1) were evaluated. Dynamic sampling period adaption was used.



(a) Average absolute cart position



(b) Average absolute pole angle



(c) Average data rate

Figure 7.14: Overview of the average absolute cart positions, pole angles, and data rates for the most relevant evaluations. Linear scale.

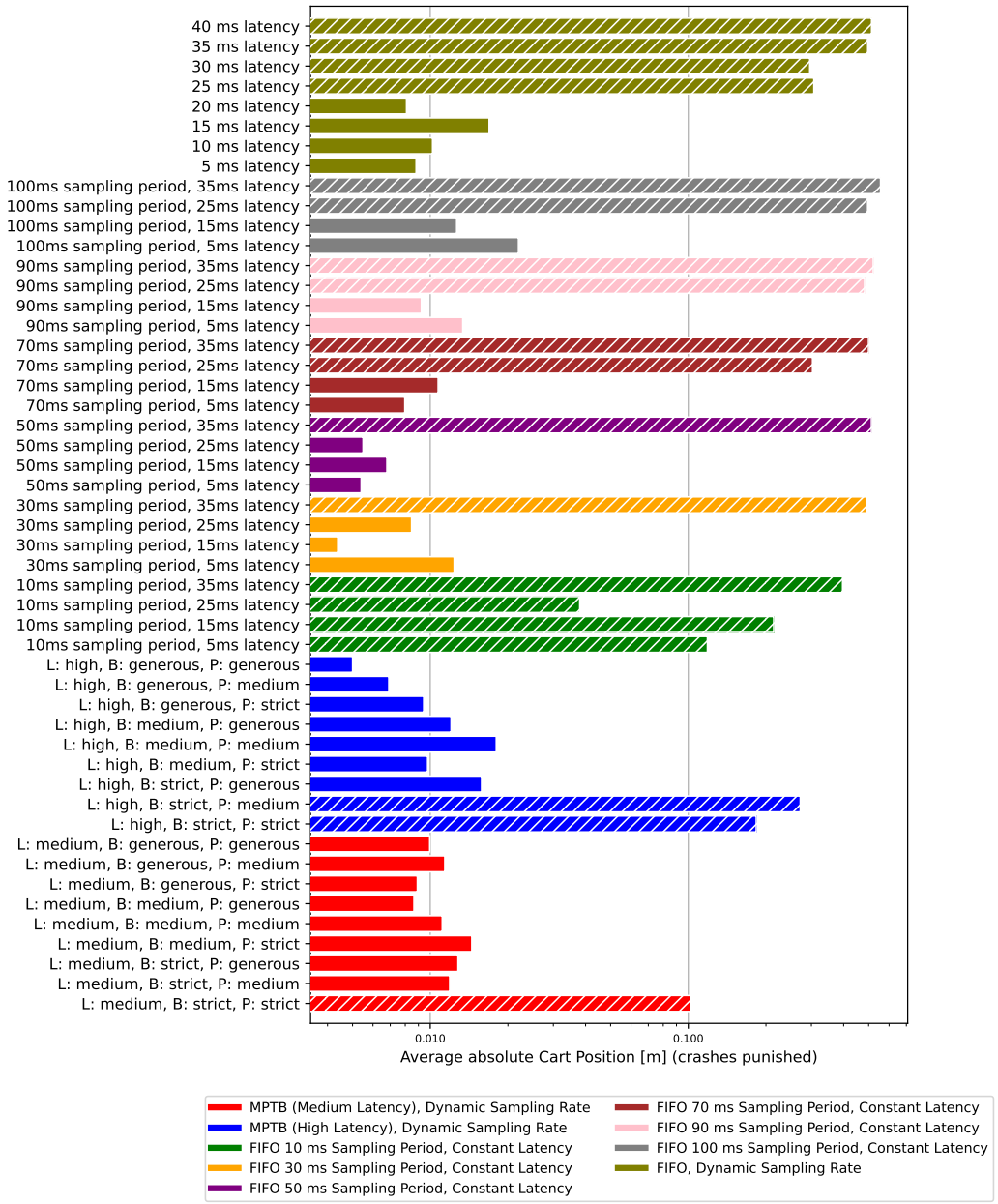


Figure 7.15: Overview plot showing the average absolute cart position over the period of the evaluation (120 s) for all evaluations with simulated delays. Evaluation runs that crashed early were punished with $\xi_{\text{cartPosition}} = 0.6$ m for the time after the crash. Crashed runs are marked with hatched bars. Logarithmic scale.

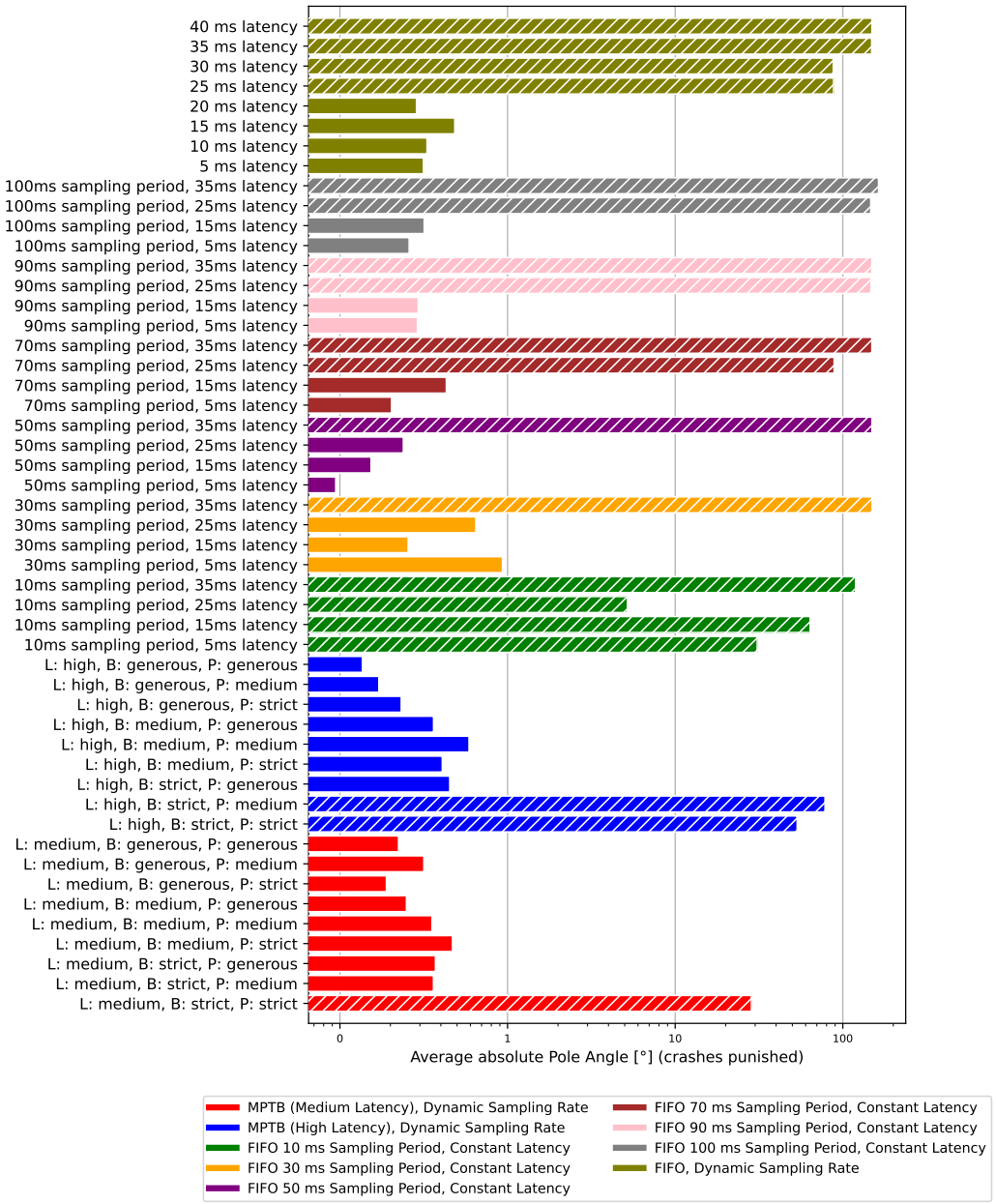


Figure 7.16: Overview plot showing the average absolute pole angle over the period of the evaluation (120 s) for all evaluations with simulated delays. Evaluation runs that crashed early were punished with $\xi_{\text{poleAngle}} = 180°$ for the time after the crash. Crashed runs are marked with hatched bars. Logarithmic scale.

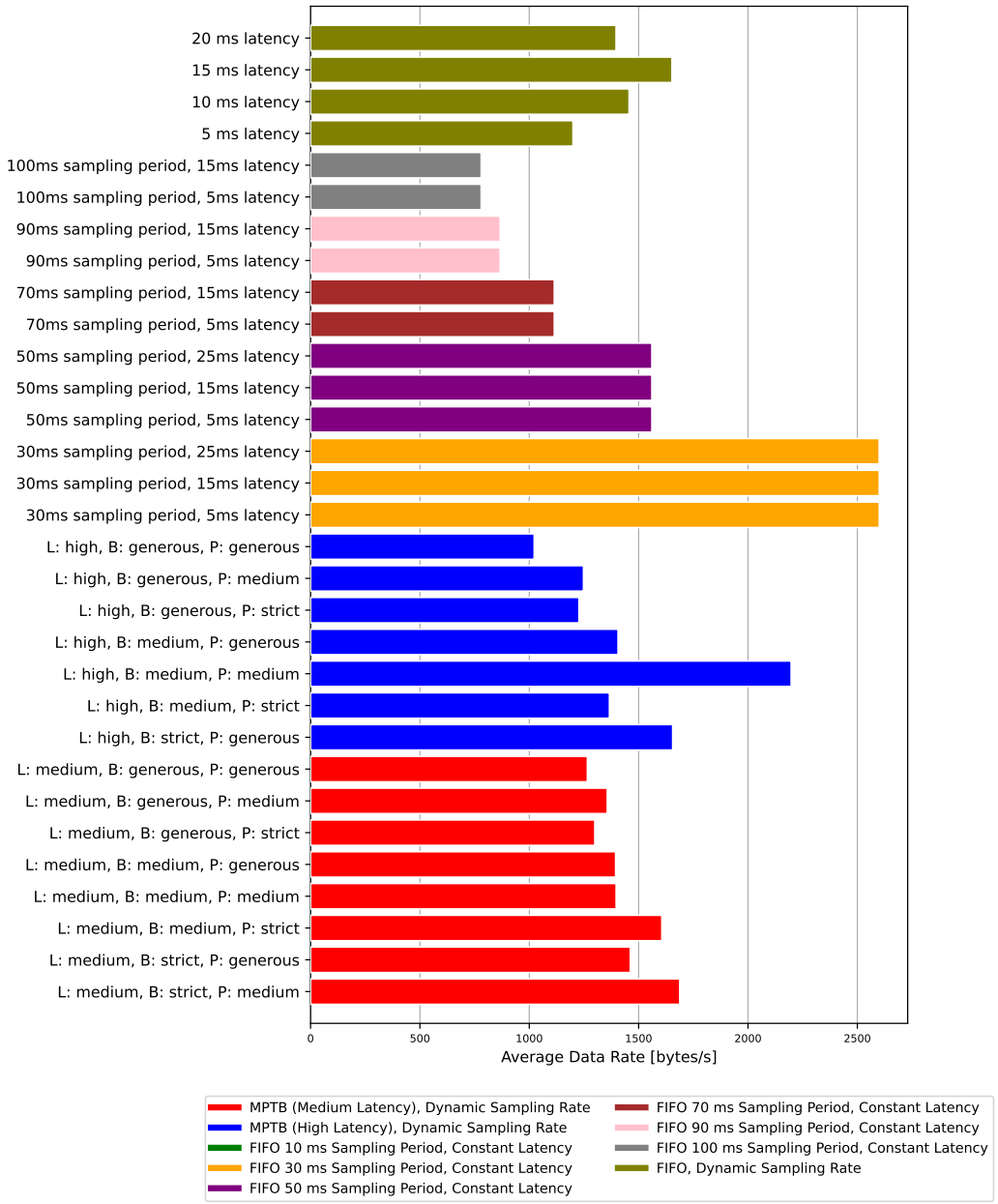


Figure 7.17: Overview plot showing the average data rate over the period of the evaluation (120 s) for all evaluations with simulated delays. Evaluation runs that crashed early were omitted from this diagram as they are incomparable. Linear scale.

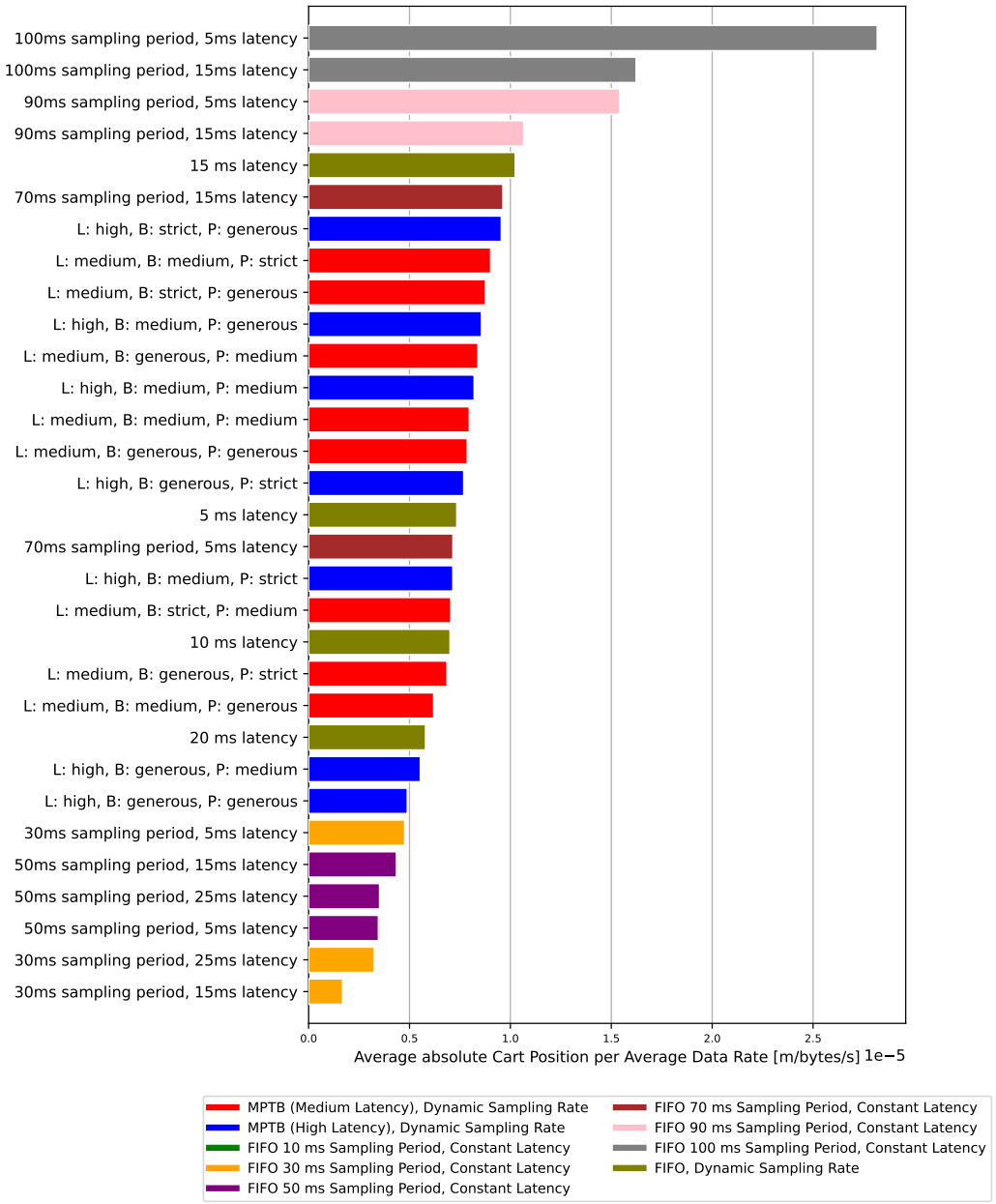


Figure 7.18: Overview plot showing the average absolute cart position divided by the average data rate over the period of the evaluation (120 s) for all evaluations with simulated delays. Evaluation runs that crashed early were omitted from this diagram as they are incomparable. Diagram is sorted for a better overview. Linear scale.

7.7 Summary of Results

In this section, we summarize the results of all evaluations we conducted.

We found that a higher sampling rate correlates with higher pendulum stability. However, this correlation is not as strong as we expected, and there is even an exception: For a sampling period of 10 ms, the pendulum becomes very unstable, while the system is the most stable for sampling periods of 30 to 50 ms. It was also found that the average absolute cart position and the average absolute pole angle have a strong correlation and can, therefore, both be used as indicators of the system's performance.

In the experiments with real cross-traffic, it was found that there is no strong correlation between priority class and latency for higher priorities. Such a correlation is present only for low priorities (≤ 3). Additionally, the latencies are generally too high to be suitable for evaluations with the inverted pendulum system, thus, rendering such evaluations useless. It was also observed that the latency severely fluctuates, making the system less predictable. Further evaluations were, therefore, only performed with simulated delays.

In the evaluations of FIFO with constant and dynamic sampling rate (CSR, DSR), it was generally found that increasing latency correlates with decreasing stability of the inverted pendulum. Most importantly, the system becomes very unstable for latencies above 30 milliseconds. It can also be observed that the probability of crashes is increased at higher sampling periods. Evaluations with a sampling period of 50 ms performed best. When comparing FIFO CSR to FIFO DSR, it was found that the dynamic sampling rate does not yield any benefit over a constant sampling rate in the scope of FIFO scheduling.

The evaluations of the MPTB scheduling algorithm show that the choice of parameters for the token bucket size and per-priority data rates have a significant impact on the performance: Strict configurations can cause the pendulum to crash due to permanently decreased priority: A disturbance causes a higher data rate, resulting in a decreased priority. The decreased priority causes a higher latency which hinders the pendulum from stabilizing, thus becoming even more unstable, resulting in an even lower priority. Eventually, the pendulum crashes. On the other hand, too generous parameters result in the pendulum's sender staying on the highest priority, even during severe disturbances. Such a choice of parameters is unnecessarily expensive in terms of bandwidth cost.

Generally, it can be observed that with increasing bucket size B and per-priority data rates P, the pendulum becomes more stable. For configurations (L_{medium}, B_{strict}, P_{medium}) and (L_{high}, B_{strict}, P_{generous}), MPTB behaves like intended: At disturbances, the priority decreases temporarily but recovers after the pendulum has stabilized again.

When comparing MPTB to FIFO, it can be observed that there are configurations where MPTB has similar or better performance than FIFO at lower average data rates and partially higher latencies. It can also be observed that the likelihood of crashes is significantly lower with MPTB compared to FIFO CSR and DSR.

8 Conclusion and Outlook

In this thesis, we presented the design and implementation of a framework to evaluate scheduling algorithms using a physical networked control system. The framework consists of a physical inverted pendulum, a sensor logic component, an actuator logic component, and an Ethernet network to transmit sensor data from the sensor logic to the actuator logic. The network contains a software switch that can run arbitrary scheduling algorithms. A logging framework to record network- and pendulum-related statistics during an experiment was developed. This system allows to evaluate arbitrary scheduling algorithms in the context of networked control systems without extensive preparation.

The Multi-priority Token Bucket scheduling algorithm, as well as a baseline FIFO scheduling algorithm were evaluated using the framework. It was found that the choice of parameter values for MPTB significantly impacts the system's performance, as too strict parameter values lead to crashes of the inverted pendulum, while too relaxed parameter values are unnecessarily expensive in terms of bandwidth. We found that, with well-chosen parameter values, MPTB can deliver higher pendulum stability than FIFO scheduling while using less bandwidth. It was also found that using real cross traffic to put a load on the switch is impractical as it results in heavily fluctuating latencies. Simulated delays at the switch were therefore used in all other evaluations.

Regarding future work, the framework constructed in this thesis can be used to evaluate other scheduling algorithms and compare their results to the MPTB and FIFO scheduling algorithms. However, also more evaluations of MPTB with different parameter values are of interest. Due to time constraints, the number of parameter values evaluated in this work was limited. Therefore, seeing whether different parameter values can give better results would be interesting.

To make evaluations easier, automating the initialization of the pendulum is also desired. Currently, a person needs to hold the pendulum's pole upwards before the pendulum starts balancing. In future work, this could be automated by rapidly moving the cart along the track to swing up the pendulum without human intervention.

Future work is also required for the use of real cross-traffic instead of simulated delays. Maybe more deterministic latency behavior can be achieved by using a dedicated programmable hardware switch, such as a P4 switch, instead of a software implementation on OS level.

A Appendix

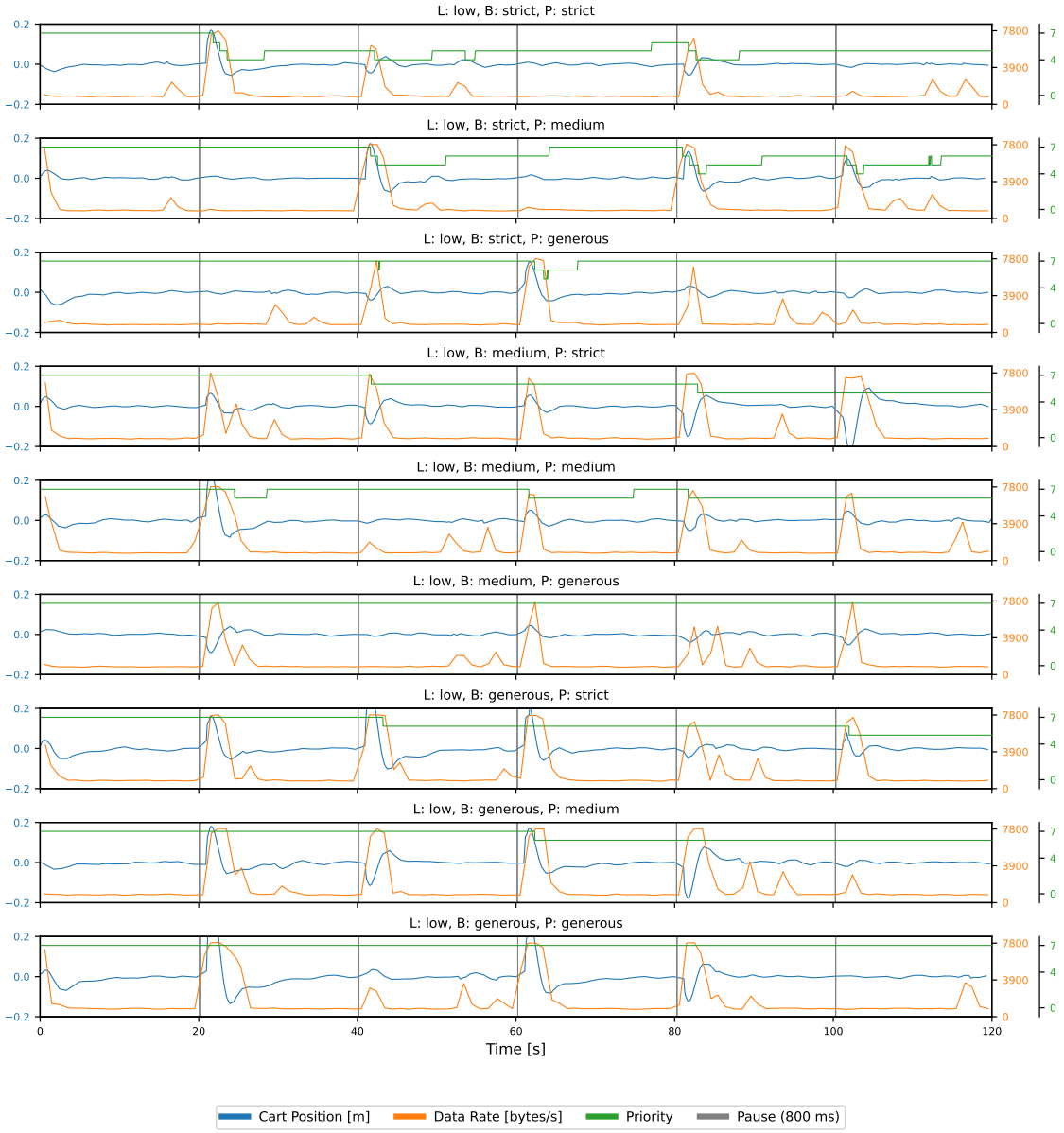


Figure A.1: Multi-priority Token Bucket evaluations with the L_{low} set of simulated latencies (see Table 7.5). All nine combinations of bucket sizes (B) (Table 7.2) and per-priority data rates (P) (Table 7.1) were evaluated. Dynamic sampling period adaption was used.

Bibliography

[Arc23] Arch Linux. *Network configuration - ArchWiki — wiki.archlinux.org*. https://wiki.archlinux.org/title/Network_configuration. 2023 (cit. on p. 65).

[Ard18] Arduino. *What is Arduino?* 2018. URL: https://www.arduino.cc/en/Guide/Introduction (cit. on pp. 21, 57).

[ARM23] ARM Limited. *ARM Cortex-A Series Programmer's Guide for ARMv7-A: Endianness*. 2023. URL: https://developer.arm.com/documentation/den0013/d/Porting/Endianness (cit. on p. 44).

[Arm23] Armbian. *ODROID C2 - Armbian*. 2023. URL: https://www.armbian.com/odroid-c2/ (cit. on p. 22).

[Ban22] Banana Pi Wiki. *Banana Pi M2 Berry*. 2022. URL: https://wiki.banana-pi.org/Banana_Pi_BPI-M2_Berry (cit. on p. 22).

[Bos17] Bosch. *Bosch XDK110 Datasheet*. Nov. 2017. URL: https://rbdevportal.secure.footprint.net/devx/dynamic_images/XDK/Guides/XDK_Node_110_combined_Datasheet.pdf (cit. on p. 30).

[Car22] B. W. Carabelli. *Performance-oriented communication concepts for networked control systems*. en. 2022. DOI: 10.18419/OPUS-12049. URL: http://elib.uni-stuttgart.de/handle/11682/12066 (cit. on pp. 17, 33, 34, 38, 39, 49, 50, 52, 61, 62).

[Edd22] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293 (cit. on p. 74).

[FHC+19] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, K. Rothermel. "NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++". In: *Proceedings of the 2019 International Conference on Networked Systems (NetSys)*. Garching b. München, Germany, Mar. 2019, pp. 1–8. DOI: 10.1109/NetSys.2019.8854500 (cit. on p. 32).

[GC08] R. A. Gupta, M.-Y. Chow. "Overview of Networked Control Systems". In: *Networked Control Systems: Theory and Applications*. Ed. by F.-Y. Wang, D. Liu. London: Springer London, 2008, pp. 1–23. ISBN: 978-1-84800-215-9. DOI: 10.1007/978-1-84800-215-9_1. URL: https://doi.org/10.1007/978-1-84800-215-9_1 (cit. on p. 19).

[GMH+20] A. Grigorjew, F. Metzger, T. Hoßfeld, J. Specht, F.-J. Götz, F. Chen, J. Schmitt. "Bounded Latency with Bridge-Local Stream Reservation and Strict Priority Queuing". In: *2020 11th International Conference on Network of the Future (NoF)*. 2020, pp. 55–63. DOI: 10.1109/NoF50125.2020.9249224 (cit. on pp. 27, 28, 35, 44).

[Her21] J. Herrmann. *Erweiterung des Mininet-Netzwerk-Emulators um einen zeitgesteuerten Scheduling-Mechanismus*. de. 2021. DOI: 10.18419/OPUS-11643. URL: http://elib.uni-stuttgart.de/handle/11682/11660 (cit. on p. 23).

[HL05] D. Hristu-Varsakelis, W. S. Levine. *Handbook of networked and embedded control systems*. TK7895. E42. H29 2005. Springer, 2005 (cit. on pp. 19, 20).

[Hub22] B. Hubert. *tc(8) - Linux manual page*. 2022. URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on pp. 24, 25).

[IEE16] IEEE. "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic". In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), pp. 1–57. DOI: 10.1109/IEEESTD.2016.8613095 (cit. on p. 32).

[IEE20] IEEE. "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34:Asynchronous Traffic Shaping". In: *IEEE Std 802.1Qcr-2020 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018, IEEE Std 802.1Qcc-2018, IEEE Std 802.1Qcy-2019, and IEEE Std 802.1Qcx-2020)* (2020), pp. 1–151. DOI: 10.1109/IEEESTD.2020.9253013 (cit. on p. 28).

[IEE22] IEEE. "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)* (2022), pp. 1–2163. DOI: 10.1109/IEEESTD.2022.10004498 (cit. on pp. 23, 45).

[KAF+15] A. Kheirkhah, D. Aschenbrenner, M. Fritscher, F. Sittner, K. Schilling. "Networked Control Systems with Application in the Industrial Tele-Robotics". In: *IFAC-PapersOnLine* 48.10 (2015). 2nd IFAC Conference on Embedded Systems, Computer Intelligence and Telematics CESCIT 2015, pp. 147–152. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2015.08.123. URL: https://www.sciencedirect.com/science/article/pii/S2405896315009908 (cit. on p. 17).

[Lin23] Linux Kernel Organization Inc. *The Linux Kernel Archives*. 2023. URL: https://www.kernel.org/ (cit. on pp. 24, 25, 68).

[Lit22] M. Litvak. *ip(8) - Linux manual page*. 2022. URL: https://linux.die.net/man/8/ip (cit. on p. 26).

[Loh23] N. Lohmann. *JSON for Modern C++*. 2023. URL: https://github.com/nlohmann/json (cit. on p. 74).

[LP22] F. Ludovici, H. P. Pfeifer. *tc-netem(8) - Linux manual page*. 2022. URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html (cit. on p. 71).

[lun23] luni64. *GitHub - luni64/TeensyStep: Fast Stepper Motor Library for Teensy boards — github.com*. https://github.com/luni64/TeensyStep. 2023 (cit. on pp. 43, 62).

[Mil22] D. Miller. *ethtool(8) - Linux manual page*. 2022. URL: https://man7.org/linux/man-pages/man8/ethtool.8.html (cit. on p. 53).

[Moz23] Mozilla. *Endianness - MDN Web Docs Glossary: Definitions of Web-related terms | MDN — developer.mozilla.org*. 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Endianness (cit. on p. 44).

[Näg15] D. Nägele. *A demonstrator for networked control systems*. en. 2015. DOI: 10.18419/OPUS-9452. URL: http://elib.uni-stuttgart.de/handle/11682/9469 (cit. on pp. 30, 34).

[Nie19] A. Nieß. *Simulation of control systems in IEEE 802.1Qbv networks using OMNeT++*. de. 2019. DOI: 10.18419/OPUS-10385. URL: http://elib.uni-stuttgart.de/handle/11682/10402 (cit. on pp. 17, 20, 32–34, 39).

[ODR23] ODROID Wiki. *ODROID C2*. 2023. URL: https://wiki.odroid.com/odroid-c2/odroid-c2 (cit. on p. 22).

[Ope13] Open Networking Foundation. *OpenFlow switch specification Version 1.4.0*. Oct. 2013. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf (cit. on p. 32).

[PJR23a] PJRC. *Teensy 3.6 Development board*. 2023. URL: https://www.pjrc.com/store/teensy36.html (cit. on pp. 20, 21).

[PJR23b] PJRC. *Teensyduino*. 2023. URL: https://www.pjrc.com/teensy/teensyduino.html (cit. on p. 21).

[PJR23c] PJRC. *Teensyduino: Using the Hardware Serial Ports*. 2023. URL: https://www.pjrc.com/teensy/td_uart.html (cit. on p. 74).

[Ras23a] Raspberry Pi Foundation. *Physical Computing with Python*. 2023. URL: https://projects.raspberrypi.org/en/projects/physical-computing/0 (cit. on p. 43).

[Ras23b] Raspberry Pi Foundation. *Raspberry Pi 2*. 2023. URL: https://www.raspberrypi.com/products/raspberry-pi-2-model-b/ (cit. on p. 21).

[Ras23c] Raspberry Pi Foundation. *Raspberry Pi Compute Module 4*. 2023. URL: https://www.raspberrypi.com/products/compute-module-4/ (cit. on p. 22).

[Rot21] K. Rothermel. *Rechnernetze I (Vorlesung)*. 2021 (cit. on p. 22).

[Sch16] B. Schirle. *Entwurf und Implementierung eines drahtlosen Orientierungssensors für ein verteiltes Regelungssystem*. de. 2016. DOI: 10.18419/OPUS-9554. URL: http://elib.uni-stuttgart.de/handle/11682/9571 (cit. on pp. 17, 30, 34).

[Sie17] S. Siegl. *Networked Control Systems: Ein Uberblick*. de. 2017. URL: https://www.unibw.de/lrt15/forschung/publikationen-1/publikationslisten/2017_07_fb_ncs_ueberblick_siegl_korrigiert.pdf (cit. on p. 19).

[Sto22] P. Stoffregen. *Teensy-Only Wiznet Ethernet Library*. 2022. URL: https://github.com/PaulStoffregen/Ethernet (cit. on p. 42).

[SWVP17] C. B. Schindler, T. Watteyne, X. Vilajosana, K. S. J. Pister. "Implementation and characterization of a multi-hop 6TiSCH network for experimental feedback control of an inverted pendulum". In: *2017 15th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*. 2017, pp. 1–8. DOI: 10.23919/WIOPT.2017.7959925 (cit. on pp. 17, 27–29, 34).

[Tan11] A. S. Tanenbaum. *Computer Networks, volume 5th ed*. 2011 (cit. on p. 22).

[Tec23] Technical University of Munich. *Networked Control*. 2023. URL: https://www.ce.cit.tum.de/en/itr/research/networked-control/ (cit. on p. 17).

[The22] The Linux Foundation. *iproute2*. 2022. URL: https://wiki.linuxfoundation.org/networking/iproute2 (cit. on p. 70).

[Thu21] P. Thubert. *An Architecture for IPv6 over the Time-Slotted Channel Hopping Mode of IEEE 802.15.4 (6TiSCH)*. RFC 9030. May 2021. DOI: 10.17487/RFC9030. URL: https://www.rfc-editor.org/info/rfc9030 (cit. on p. 28).

[Weh02] K. Wehrle, ed. *Linux-Netzwerkarchitektur: Design und Implementierung von Netzwerkprotokollen im Linux-Kern ; [zu Kern 2.4]*. Linux Specials. München ; Boston[u.a.]: Addison-Wesley, 2002. ISBN: 3827315093 (cit. on p. 24).

[Wik12] Wikimedia Commons / Krishnavedala. *Schematic drawing of an inverted pendulum on a cart*. 2012. URL: https://commons.wikimedia.org/wiki/File:Cart-pendulum.svg (cit. on pp. 20, 37, 43).

[ZHG+20] X.-M. Zhang, Q.-L. Han, X. Ge, D. Ding, L. Ding, D. Yue, C. Peng. "Networked control systems: a survey of trends and techniques". In: *IEEE/CAA Journal of Automatica Sinica* 7.1 (2020), pp. 1–17. DOI: 10.1109/JAS.2019.1911651 (cit. on p. 39).

[Zin16] S. Zinkler. *In-network packet priority adaptation for networked control systems*. en. 2016. DOI: 10.18419/OPUS-9401. URL: http://elib.uni-stuttgart.de/handle/11682/9418 (cit. on pp. 17, 31, 32, 34, 39).

All links were last followed on March 14, 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