Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Flow Control for Event-based Cyber-physical Systems

Tom Rohloff

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Ing. Ben Carabelli
Commenced:	2018-04-09
Completed:	2018-10-09

Abstract

Cyber-physical systems (CPS) gained widespread attention in recent years. In the context of Industry 4.0, they are often implemented as Networked Control Systems (NCS) that use a shared network. Past research already investigated stability, as well standard performance criteria of the control system. Based on different trigger functions of previous work, the question is addressed how their trigger conditions should be parameterized to obtain optimal results with respect to the performance. This thesis focuses on discrete-time Linear Quadratic Regulators (LQR) that operate in bandwidth-limited networks. Leveraging the quadratic nature of the problem, the optimization technique proposed uses gradient descent and heuristics based on other metrics gathered from the network. To evaluate this approach, a simulation framework is developed. The quantitative analysis shows that a proposed trigger parameter adoption policy performs very well. The overall control system performance is only marginally worse than the optimum (about 4 % on average). Compared to the naive approach of triggering at every sampling instant, performance gains ranging from 6 % to over 500 % (depending on the bandwidth limit) could be achieved. Additionally, the control system performance stays consistent, even in the presence of extreme constraints of the relative link capacity.

Contents

List of Figures

List of Abbreviations

CPS Cyber-physical Systems. 11

GCC GNU Compiler Collection. 45

GPU Graphic Processing Units. 49

GUI Graphical User Interface. 42

IDE Integrated Development Environment. 45

IoT Internet of Things. 11

LQR Linear Quadratic Regulator. 21

NCS Networked Control System. 11

NW-JSP No-wait Job-shop Scheduling Problem. 15

ODE ordinary differential equation. 19

PRNG pseudo-random number generator. 38

QoS quality of service. 15

TSN Time-sensitive Networking. 15

1 Introduction and Motivation

Recent developments in computer science, electrical and control engineering over the last few years led to the emergence of a field of study called Cyber-physical Systems (CPS). Such systems consist of software components combined with physical elements such as mechanical or electronic parts. These components communicate over a network infrastructure, which allows for highly flexible settings regarding properties like autonomy, scalability or robustness [Lee08].

CPS are deemed as a critical (i.e., necessary) component of Industry 4.0. This term describes an organization principle of modern manufacturing processes that is composed of four basic concepts [HPO16]:

Interconnection Machines, sensors, devices and people are able to communicate with each other via a communication network, e.g., over the Internet of Things (IoT).

Information transparency The aforementioned interconnection enables operators to access a vast amount of information on possibly all aspects of the manufacturing process, thereby allowing and furthering its improvement.

Technical assistance This aspect is twofold. First, gathered information is aggregated and visualized in a way that allows for easy analysis and well-founded decisions. Second, the CPS supports human personnel in exhaustive, dangerous or otherwise unpleasant tasks.

Decentralized decisions The CPS is able to make decisions and perform its tasks as autonomously as possible. Only in the case of an emergency or goal conflict, the system hands over its current task to a higher authority such as an operator.

CPS constitute a primary building block used to implement this organization principle. While aspects like interconnection and decentralized decisions state *what* is to be achieved, a CPS describes *how* it can be realized. The combination of hardware and software components in these systems often takes the form of some physical (i.e., mechanical or electronic) devices that are controlled, supervised or otherwise connected to a "cyber" (i.e., software) system.

This non-physical part computes the action that is to be performed based on some input measured by the physical part. The input is transmitted over the interconnection network to the software system, transformed into some instructions and sent back to the physical part via the same network. From a logical or architectural point of view, these connected components are often seen as a single device. From the implementation's standpoint, however, it is perfectly reasonable to implement them as separate but connected components in order to fulfill and improve non-functional requirements like extensibility or fault-tolerance.

A possible scenario with CPS is sketched in Figure 1.1: Multiple sensors measure various metrics like the temperature and pressure of a given physical process like a chemical reaction in a reactor. The measured values are transmitted over the network to the software component which then computes (based on this information) a response action that is sent to and executed by actuators. These may alter some properties (e.g., lower the temperature in the reactor) that influence the physical process.

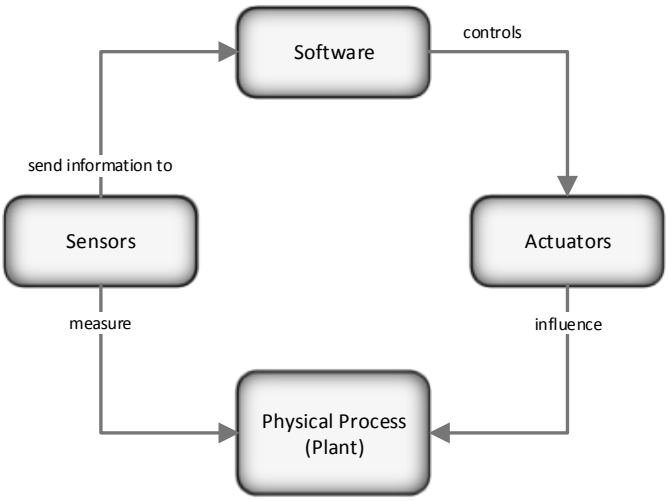

Figure 1.1: The basic principle of a Cyber-physical System.

Two key aspects of CPS are outlined in this scenario. First, the process could be monitored and adapted continuously. This means that information that is measured at one point in time is "fed back" into the physical process and therefore influences the information gathered the next time a measurement is performed. These are essentially the characteristics of a feedback control loop. The whole topic of control systems is covered in great detail in Section 3.1. Second, the communication happens over a possibly shared network. This means that the transmission of information is not accomplished by point-to-point communication lines, but through a packet-switched network shared by multiple participants. This results in the fact that not the whole bandwidth may be available at all time instants to transfer all the information needed to obtain an optimal behavior of the physical process. Furthermore, the packet switching enforces a discrete (in contrast to a continuous) view of time since data could be transmitted in chunks only. This leads to a minimum period of time between two packets (depending on the communication protocols used and bandwidth available) and thus to a discretization of time. The CPS in this scenario could therefore be seen as a Networked Control System (NCS), since its underlying mechanism resembles a control system, and it uses the network.

Although the primary goal of a NCS is mostly to maintain stability of the controlled process (which ensures safety), another important objective is to maximize the efficiency, i.e., to get a better *performance* of the system. This is done by defining a cost function that describes the output of the physical process, which is then sought to be minimized. A common observation for control systems is, depending on the specific discretization and system dynamics, that the performance is not affected greatly if the controller does not complete its control cycle at every possible point in time. Transferred to NCS, this may allow a vast reduction of the bandwidth required, while maintaining a relatively good performance. The point in time at which a packet is transmitted is determined by evaluating a "trigger condition" internal to a *trigger function*, which considers parameters related to the NCS' such as the internal state, performance or traffic pattern.

Even though multiple trigger conditions have been proposed, it still remains an open question how their parameters should be properly parameterized in order to find the optimum between two extrema: First, the case in which the packet is sent at every time step. This leads to a high network

load and possibly higher costs since the network capacity available for NCS traffic is assumed to be constrained, resulting in more packet drops, which may degrade the performance. Second, the trigger function is dimensioned such that too few no packets are sent. This minimizes the network load, but it, too, degrades performance since too little information is fed back to the controlled process.

The goal of this Master Thesis is the development of a method that allows for a dynamic adaption of these trigger parameters in a way that uses the available network slice optimally while keeping the costs at a minimum (i.e., maximizes the performance of the control system).

1.0.1 Outline

The remainder of this thesis is structured as follows: In Chapter 2, previous research on CPS is covered, in particular those investigations directly related to the contribution of this thesis. This includes approaches for performance optimization happening in both the network itself and at each NCS individually. Chapter 3 covers the fundamentals required for the understanding of the following chapters. A system model is introduced in Section 4.1. Based on this foundation, several trigger adaption policies are described in Section 4.3.3. This thesis includes the development of a software used to simulate the proposed policies. Details on this implementation are given in Chapter 5. With the results obtained with the software, an evaluation of the policies follows in Chapter 6. In chapter 7, this thesis is concluded and future work presented. Appendix A contains all information on how to download, use and extend the software developed.

2 Related Work

In order to maintain stability of a Networked Control System, the connecting network needs to satisfy (among others) real-time constraints. Given this property, it is possible to use the low and bounded latency to continuously control the process in question using a feedback control loop. While this is accomplished relatively easily with dedicated networks (such as CAN bus) that yield a guaranteed service, maintaining real-time properties is a notoriously difficult task if the network is shared. This leads to the question, how control-specific quality of service (QoS) guarantees could be implemented in general-purpose networks (e.g., those networks that provide only best-effort within a certain traffic class).

Dürr and Nayak recently proposed an approach leveraging the Time-sensitive Networking (TSN) extensions to Ethernet (referred to as IEEE 802.1Qbv) with the goal of supporting multiple NCS [DN16]. The idea is to allocate exclusive and isolated resources to single NCS. This is done by the assignment of reserved time-slots to individual flows through a "gating" mechanism and possibly traffic shaping. The gating mechanism is essentially a packet scheduling policy implemented in the network infrastructure such as switches. When a gate is opened for a specific flow, only packets belonging to this flow are scheduled, while all others are held back (i.e., queued or even dropped). By synchronizing the opening and closing of a flow's gate at every switch involved, very low latencies could be achieved and (within some bounds) even guaranteed. The author's approach now improves the throughput and enables lower latencies by solving the No-wait Job-shop Scheduling Problem (NW-JSP). The basic idea of this technique is to minimize bandwidth wastage by rearranging the flows scheduled in a switch such that the amount of gate openings is reduced. Thus, fewer of so-called "guard bands" are needed. A guard band is essentially the time required to serialize a packet, i.e., to put it on the communication line. Considering the scheduling of time-critical traffic, it is inserted just before the gate of a specific flow is opened in order to ensure that no best-effort traffic packet is still in the process of being transmitted (which would otherwise delay the opening of the flow-specific gate). Essentially, this separates the best-effort traffic from time-critical traffic.

The gating mechanism is complemented by traffic shaping techniques like token bucket or leaky bucket. For these methods, see [Tan+03].

While this constitutes a promising approach regarding the individual guarantees of a NCS, it comes with the downside of some traffic and processing overhead that occurs within the network. In this setting, NCS constantly transmit information while executing the control loop resulting in periodic traffic that must be handled. This may lead to higher drop rates of unrelated best-effort packets, and therefore to an increased load on the network which may be avoidable or at least reducible in comparison to the gained performance. Another possible drawback could be seen in the way priorities are assigned to the respective network flows. This is handled in a static manner which means that no adjustment "at runtime" is possible. Once the resources (i.e., isolated network slices) are reserved and the priorities set, there is no way to adapt this setting dynamically to varying network loads or numbers of NCS. Such a situation may occur with control systems if the transmission period must be changed. This would require a re-computation of the schedules by solving the NW-JSP for the new setting.

An enhancement to this approach is suggested by Carabelli, Blind, Dürr and Rothermel in [CBDR17]. The aforementioned lack of dynamic adaption is remedied by computing the priorities not once beforehand, but at the sensors themselves with a state-based method. By dynamically adapting the priorities at each NCS individually, a stateless priority queuing in the network is possible. This technique enables it to fully utilize the network slice reserved, which in turn is optimal with respect to the performance of the NCS group.

Both of these two approaches suffer from the constraint that the scheduling itself must be implemented and done in the network which may amount to a significant cost disadvantage or implementation overhead.

An alternative and possibly improvement is *event-based control* (see [Ast08]), which leverages the observation that a control system's performance may not suffer greatly if a packet is not transmitted at every point in time. Instead, the time of transmission is selected based on some internal state of the sensor, which is computed using a *trigger function* that takes various metrics gathered from the network and sensor itself into consideration. This takes the burden of scheduling/processing from the network itself and shifts it to the CPS component that transmits the packet (i.e., the sensor). Furthermore, it does not rely on Ethernet extensions like IEEE 802.1Qbv. While previous approaches focus on the stability of the control system, Linsenmayer and Allgöwer concentrate on the system's performance [LA18]. This vastly reduces the amount of packets since periodic traffic from the CPS is almost eliminated.

In this thesis, the stability of the system is disregarded and only opportunistic (i.e., best-effort) transmission guarantees are assumed. Conceptually, it is possible to combine both approaches by using periodic transmissions and event-based triggering simultaneously to make guarantees about the stability and improve the performance. For an exploration of this subject, see the work of Martí, Velasco and Gaid in [MCVG10].

3 Fundamentals

The following sections discuss all the basic background information needed to understand the concept presented in Chapter 4. First, the various types of control systems are introduced, with an emphasis on closed-loop feedback control and Linear Quadratic Regulators, since such a system is used at the core of the concept and simulation. Next, different triggering methods are presented, namely the time-based trigger, event-based trigger and self-triggered control. A quick discourse about optimization and its methods like converging or heuristic algorithms follows.

3.1 Control Systems

Systems that change over time are virtually everywhere the real world. Modeling them regarding the system's state allows a relatively easy handling in mathematical sense. There exist countless examples, ranging from fluid flow over a wing, the population dynamics of a city, the spread of a disease or a model of the planets moving around the sun in a solar system. The main advantage of such a model is the ability to predict future states of the system in question by looking at the current state.

Taking this idea one step further leads to the manipulation of systems in order to change their behavior. There are two distinctive ways to accomplish said change: First, use an input to the system and manipulate it through this interface. Second, use not only an input, but measure some metrics of the system and act according to the measured values. Then it is possible to design a control policy to change the behavior based on the measurements by changing the input to the system.
Two main types are distinguishable:

Passive control This approach uses no active manipulation of the system's state, but instead relies on predefined (i.e., engineered) interrelationships between an input to the system and a desired output. An example for such a basic system is the (statically mounted, non-retractable) spoiler on a sports car. The input corresponds velocity of the car, which is not modified by the control system in any way. This velocity relates to an air flow that is guided by the spoiler in a way that exerts a force pointing downwards to the ground, resulting in more static friction of the tires touching the ground. Therefore the car may sustain higher lateral accelerations without losing grip (i.e., without over-/under-steering). The relationship of input (velocity) and output (force) is set once when the spoiler is designed. It's not adapted or otherwise modified during its use. Often such a passive system yields insufficient performance, which leads to active control.

Active control Using active control it is possible to change the way the system behaves. There are two subtypes of control systems:

Open loop This type of control system uses an input that is used in the controller to compute an output. This output is then applied to the underlying process. In general, the system is reverse-designed in order to get an abstract model. Then, a trajectory (i.e., input corresponding to the desired output) is preplanned, and afterwards enacted as control

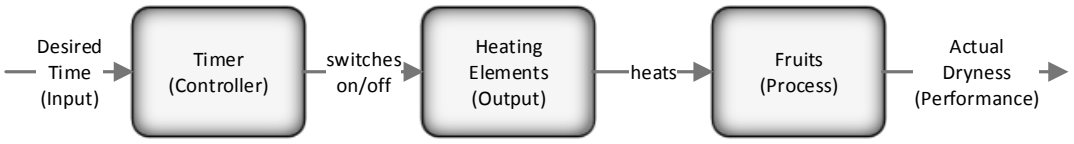

Figure 3.1: Principle of an open-loop control system.

law. The system may become unstable if the input is not applied continuously. There is no measurement and therefore no feedback of the output or the current condition of the process in question, hence it's called an open loop. This prevents an adaption of the input in response to a changed environment. More specifically it allows no reaction to disturbances, e.g., external noise sources. This possibly results in a worse performance of the system.

Figure 3.1 illustrates an example of this principle. The overall goal and underlying process in this setting is to dry fruits by applying heat. The desired time over which the fruits are dried serves as input, while the controller takes the form of a timer that switches the heater elements (output) on or off. From this point of view, the performance is the actual dryness of the fruits: The dryer the fruits, the better the performance.

A disturbance may come into play in terms of a change that may reduce the system's ability to complete the desired task. For example, the dryer box's hatch opens and the required heat dissipates. The open loop control system has no way to detect this change and the timing controller continues to power the heating elements for the remaining time without noticing or at least alerting an operator.

Closed loop feedback control In contrast to the open loop approach above, the closed loop feedback control essentially makes use of sensors that measure what the system is actually doing (i.e., measuring its output) and then computing an input based on these measured values and the desired output. The output is therefore "fed back" into the system. Compared to open loop systems, the feedback allows an improvement in the following four main aspects:

- Uncertainty, for example an underlying process that does not meet the assumed specifications (i.e., modeling errors) exactly or noisy measurements. All these uncertainties could be handled with feedback by measuring the deviation of the actual system from the predicted one, and react accordingly.
- Disturbances, i.e., sudden changes to the system that may be impossible to predict. These are external to the system, for example a rising temperature of the environment, wind blowing, radiation and so forth.
- Stability. The underlying process may be unstable, which means that the system's state tends to drift unavoidably away from a well defined optimum or desired state. For some systems though, it is possible to use feedback in order to stabilize the system. This aspect is addressed in greater detail in the following paragraph.

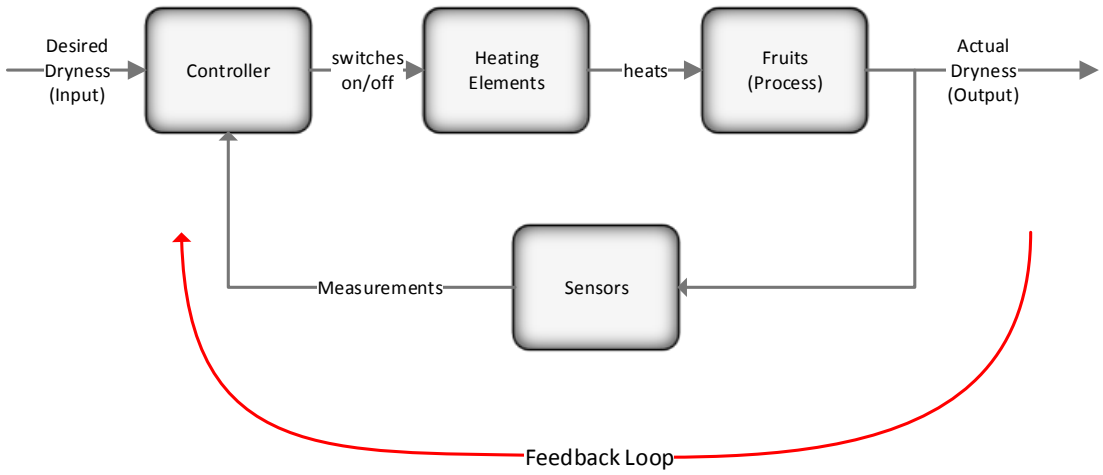

Figure 3.2: Principle of a closed-loop control system

- Efficiency. Since the actual output is measured, it is possible to optimize the input in a way that improves the performance of the system (i.e., reduces the costs). Usually a cost function is defined as a measure of the control performance, and then minimized. This is exactly the approach used in this thesis to evaluate and optimize the trigger functions and their parameter adaptions.

Figure 3.2 shows the previous example of a fruit drying machine as a closed loop feedback control.

Here, the input is no longer a time value (as it was in the open loop example), but a desired dryness. The controller computes the remaining time necessary the heating elements are switched on to reach the required dryness. The sensors in turn measure the current temperature and humidity, and transmits these values to the controller. With the measurements, the controller is able to compute (or estimate) the current dryness, compares it with the desired dryness (i.e., computes an error), and adapt the time accordingly. This closes the feedback loop. If a disturbance occurs (such as an open hatch), the system may detect this and either adapt the remaining time, close the hatch or inform an operator of an exceptional situation it cannot resolve itself.
This thesis focuses on closed loop feedback control systems only.

Systems like the one described above and shown in Figure 3.2 are often modeled as a linear system of ordinary differential equation (ODE)s. The basic form is

$$\dot{x} = Ax, \tag{3.1}$$

with the vector $x \in R^n$ that contains all the quantities of interest in the system, thus representing the current state of the system. The matrix A, also called the system matrix, describes how x evolves over time. A solution to equation 3.1 is

$$x(t) = e^{At}x(0), \tag{3.2}$$

where $x(0)$ is the initial state of the system (i.e., at time point 0). Furthermore, control theory states that if the matrix A has any eigenvalues with a positive real part, the system will be unstable, i.e., $|x|$ grows exponentially for any non-zero initial state $x \neq 0$. If all of the eigenvalues have negative real parts though, then the system has stable dynamics and they converge to zero as time goes to infinity. In order to control the system, a second matrix B is added to equation 3.1:

$$\dot{x} = Ax + Bu, \tag{3.3}$$

with u serving as a kind of "control knob" that is used to manipulate and stabilize the system, thus u represents the input. B in turn describes how this manipulation directly affects the time rate of change $\dot{x}$ of the state x.

On the Instability of a System

The manipulation of the system via u can be used to react on the current state x. By choosing u based on the control law K as $u = -Kx$ and inserting it in the original equation of a control system (eq. 3.3), this results in:

$$\begin{aligned} \dot{x} &= Ax - BKx \\ &= (A - BK)x \end{aligned} \tag{3.4}$$

The equation 3.4 above shows that by measuring the (full) state x and feeding it back to the control input u (through $u = -Kx$), it is possible to change the dynamic matrix so that a new dynamic system $\dot{x} = (A - BK)x$ (in contrast to the one from equation 3.1) comes into existence. The stability of the system now depends on the eigenvalues of $(A - BK)$, which makes it possible to asymptotically stabilize an originally unstable system by designing the matrix K accordingly.

Example: Inverted Pendulum

To get a better grasp of the abstract concept described above, the following example is presented. Figure 3.3 shows a cart that could be moved horizontally. On this cart, an inverted pendulum is mounted. The goal in this setting is to stabilize the pendulum with its end facing upwards in such a way that it doesn't fall over, as it normally would.

The stabilization is then accomplished by measuring the current state of the system and moving the cart with different speeds either to the left or to the right. By defining the state of the system as a vector of states x being

$$x = \begin{bmatrix} p \\ \dot{p} \\ \alpha \\ \dot{\alpha} \end{bmatrix}, \tag{3.5}$$

where p is the current position of the cart, $\dot{p}$ is the cart's velocity (i.e., the derivative of its position), α is the angle by which the pendulum deviates from the optimum (upright) position and $\dot{\alpha}$ is the angular velocity of the pendulum. Together, these form a system with two degrees of freedom.

This non-linear system $\dot{x} = f(x, u)$ must then be linearized around a fixed point. To do this, a fixed point must be found first. A fixed point is one x from the state space $\mathbb{R}^n$ where the state of the system does not change. Intuitively, this is the case if neither the cart, nor the pendulum moves at

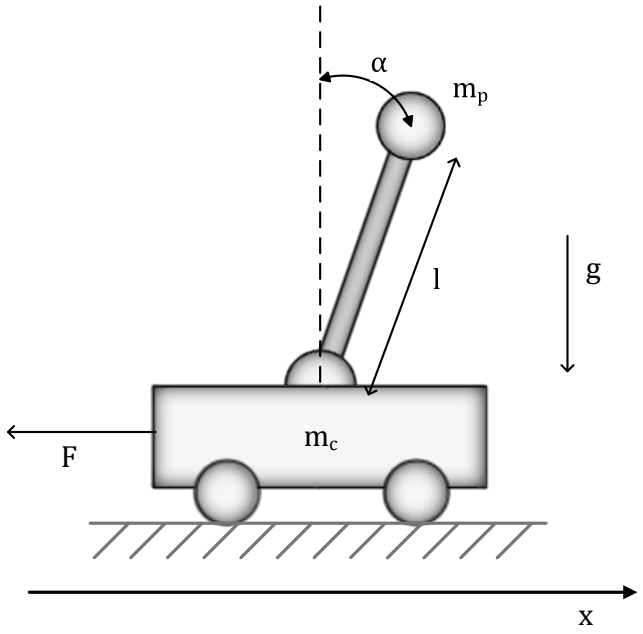

Figure 3.3: An inverted pendulum and its essential properties.

all. In this scenario, this happens if the pendulum is either exactly in its upright position, or if it is facing downwards. This leads to a linearization such that

$$\begin{aligned} A &= \frac{\partial f}{\partial x}(0) \\ B &= \frac{\partial f}{\partial u}(0). \end{aligned} \tag{3.6}$$

After linearization, one gets a linear system of equations with matrices A and B: $\dot{x} = Ax + Bu$ (eq. 3.3). The control input to the system u then equals a force on the cart that pushes it in the x-direction. To complete the description of the system, more parameters must be considered, such as the length of the pendulum l, its mass m_p and the mass of the cart m_c, and the gravity g that pulls the pendulum downwards. In the case of a simulation, a friction (i.e., damping) should be included as well, possibly applied to the cart's velocity.

3.1.1 Linear Quadratic Regulator

With equations 3.3 and choosing the control law K as $u = -Kx$ (eq. 3.4), the matrix K could be computed so that the system will be driven to some specific eigenvalues. This leaves the question how to determine the best eigenvalues, and how to know if the current ones are the best.

The answer lies in the Linear Quadratic Regulator (LQR). Its basic idea is to craft a cost function that considers both the deviation of the current state from the target state, and the costs of the regulation itself. The basic form of such a cost (i.e., performance) function J is as follows:

$$J = \int_0^\infty (x^\intercal Q x + u^\intercal R u) dt, \tag{3.7}$$

where x and u are both functions of the time t, which is why the sum is integrated with respect to t.

The two summands are penalties of the current state and the regulation costs for this state. Q describes how much of a penalty is added for the state deviation, while R characterizes the penalty associated with the regulation. Intuitively, by integrating with respect to the time t, the total costs rise if either the state is not stabilized quickly enough, or by doing so a lot of energy is consumed.

By designing the matrices Q and R carefully, it is possible to weight the different aspects of the system. Following the example of the inverted pendulum above with a four element state vector x, the matrix

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix} \tag{3.8}$$

would penalize the position x and velocity $\dot{x}$ of the cart with a factor of 1. In contrast, the higher penalty values for the angle of the pendulum α and its angular velocity $\dot{\alpha}$ mean that a deviation from the upright position is far more grave than a deviation of the cart's position. Therefore, the system would try to stabilize the angle very quickly.

With these known penalty matrices Q and R there exists a best control law, i.e., an optimal K matrix for a given system that minimizes the cost function. For details on its calculation, refer to [LA18].

This controller K is the Linear Quadratic Regulator. It uses a linear matrix K (multiplied with the current state, eq. 3.4), minimizes a quadratic cost function J (with a well defined minimum, eq. 3.7), and stabilizes the system (i.e., the state x will go to zero, eq. 3.4). In contrast to other controller approaches like PID (see [PM02]), the LQR is by definition the best controller achievable with respect to the Q and R matrix.

3.2 Discrete Time Control Paradigms

Recent developments led to the emergence of NCS. Such a system is basically a control system that is physically distributed. The building blocks of a NCS such as the sensors, the controller itself or the actuators are connected over a (possibly shared) network. Although this decoupling from the physical location yields many advantages like high extensibility and robustness, it comes at the costs of a relatively high network load. At every time instant, both all the information gathered from the system (i.e., the measurements of the output), as well as the computed input must be transmitted over the network in order to close the feedback loop. With potentially many NCS in use at the same time, the amount of generated traffic may exceed the network's link capacity easily, especially if a wireless network connection with a drastically reduced bandwidth (in contrast to wire-bound networks) is used.

In order to reduce the traffic generated by multiple NCS, the following observation is leveraged: The performance of a given control system does not suffer greatly if the output is not fed back at every time step. This led to the development of various so-called "trigger" methods. Such a method determines (possibly based on an internal state of the trigger) whether the information should be transmitted at the current time instant or not. In the following, three established methods are briefly described. For more information on this subject, refer to [HJT12].

3.2.1 Periodic Time-based Triggering

When implementing a controller in digital hardware, the time cannot be viewed as continuous, but must be discretized. Consequently, a control system is regarded as a sequence of states, and a transition from one state to the other happens only with the discretization interval period. Note that in addition to the limitations imposed by the (finite) digital representation of values, the use of a packet switched network leads to a discretization, too. This is due to the splitting of information into discrete chunks of data that form the packets sent over the network.

With a discretization of time given as

$$\begin{aligned} x_k &= x(t_k) \\ t_{k+1} &= t_k + T, \end{aligned} \tag{3.9}$$

where T is the period of time between two sampling points in time, a discrete time system, which may have a continuous underlying system, could therefore be modeled as

$$x_{k+1} = Ax_k + Bu_k. \tag{3.10}$$

The periodic time-based triggering method is the most simple form of triggering. Instead of transmitting the information continuously (which is not feasible in packet-switched networks anyway), it is sent over the network only at every sampling time t_k.

While this approach allows for a very easy implementation, it is often not appropriate in regard to the control system's performance. Since the current system state is not considered it may be unsuitable for situations in which, for example, a lot of disturbances occur. For instance, a transmission happens at time $t = 10\ ms$, the period for transmissions is $T = 10\ ms$. Just after that last transmission, a massive but normally controllable change of state occurs, i.e., at time $t = 11\ ms$. Even if the system's sensors could measure the change at $t = 11\ ms$, it cannot be compensated by regulating the input until $t = 20\ ms$, which means that over 9 milliseconds are wasted in this scenario.

Another problematic situation could arise in environments where a lot of packet drops occur, such as in wireless networks. Closely related to the scenario above, the disturbance happens at time $t = 10\ ms$. The sensor measures this and a new input is computed to accommodate the system accordingly. Given that the packet with this input is sent but not received at the actuator due to a packet drop, the performance is reduced for at least 10 milliseconds.

In both cases, the control system cannot react to the change of state or the loss of a packet accordingly, since the only possibility to adapt is at the next point in time when a transmission is scheduled. This led to the development of another approach called "Event-based triggering".

3.2.2 Event-based Triggering

The disadvantages of the previous, periodic time-based method could be compensated by transmitting not at a static interval, but by adjusting the point of time a packet is sent over the network continuously. This is accomplished by making the transmission decision based on an internal trigger state, allowing to consider not only the current state of the system, in particular the stage costs (i.e., performance of the current step), but also the history of previous trigger decisions.

The simplest event-based trigger policy compares the current penalty with a threshold, and transmits the packet at the time instant the limit is exceeded. This trigger condition c is monitored continuously, i.e.,

$$t\colon\ c(x(t)) = 0; \quad t \in \mathbb{R}_+ \tag{3.11}$$

In the scenario sketched above, this would solve the situation and keep the costs low, since a retransmission would be scheduled immediately after the disturbance happens or the packet is lost, because the rising penalty would be detected.

3.2.3 Self-triggered Control

The Self-triggered control approach was introduced by Velasco, Fuertes and Marti in the context of host-local (in contrast to networked) real-time systems [VFM03]. In Event-based triggered systems, the output is monitored continuously. Some control systems suffer from the fact that such a continuous measurement is infeasible. The basic idea of the Self-triggered policy is that the controller computes not only the next input for the actuator, but also the next time instant at which the control law is recomputed (i.e., the "trigger" instant). This information is then relayed to the scheduler.

Depending on the control system in question, this approach can reduce the resource utilization (i.e., used network bandwidth or computing time) since the information measured by the sensor must not be gathered and transmitted continuously to the (possibly) physically distributed controller. Furthermore, the Self-triggered control method may fit if the sensors are not able to keep an internal state, e.g., because of memory limitations.

3.2.4 Periodic Event-based Triggering

This policy combines elements of both the Event-based triggering, and the periodic Time-based triggering. Now, the trigger condition c is not evaluated continuously, but at discrete time instants with the period T apart from each other, i.e.,

$$t_k : \; c(x(t_k)) = 0; \quad t \in \{nT\},\; n \in \mathbb{N}_0 \tag{3.12}$$

With this approach it is possible to craft a trigger policy that not only allows a quick reaction to external influences like disturbances, but in addition it makes it possible to minimize the network traffic caused by a the NCS in question. In the scenario above, the sampling period T can be reduced to $T = 1\ ms$. The change of state would then be detected and the packet transmitted at time $t = 11\ ms = t_{11}$.

Trigger policies could be designed with the stability of the system as primary goal. A different approach is investigated by Linsenmayer and Allgöwer in [LA18]. The authors focus on various trigger policies that keep the performance of the control system well in bounds, using periodic event-based control, where triggering is also restricted to discrete time instants.

In order to take both stability and cost limits into account, it is possible to combine the periodic Time-based and Event-based triggering approaches in such a way that both periodic and event-triggered transmissions occur. This is further discussed in [MCVG10].

3.3 Optimization

The term "optimum" describes the best result achievable by a system in the sense of a trade-off between various parameters or properties. The act of optimization therefore describes the search for parameters that result in an optimum of the so-called target function f that uses those values. In a mathematical context, optimization is the search for extrema, i.e., minima or maxima of f.

The classic approach is the analytical investigation of the target function, which typically entails the computation of the first derivative (i.e., f') and requires f to be continuously differentiable. While this procedure is easily conducted for some problems, there exists a multitude of cases where the direct, analytical approach with desirable finitely terminating algorithms is infeasible. Possible reasons include:

- The problem has a combinatorial nature, e.g., the knapsack problem.
- There is no closed-form solution (i.e., an expression that can be evaluated in a finite number of steps) as it is the case with the Three-body problem.
- The solution is not computable within acceptable time or memory boundaries.
- The computation of the solution is afflicted with numerical errors that accumulate quickly (as it happens with power series like the geometric series). These errors arise most often from the discretization of real numbers, since today's computers can only process finite precision values.
- The function f is not known at all, but some some characteristics such as the existence of an extremum or the degree of a polynomial could be assumed. This may be the case because the locally accessible knowledge about f is not be sufficient to use an analytical approach and compute the extremum at once.

In such situations, convergent iterative methods that do not suffer from the above limitations are used instead. Such iterative methods generally do not terminate after a specific amount of steps, but they improve the quality (i.e., mostly the precision) of the computed solution in every iteration, which lets the error converge to a low constant value, desirably zero. They therefore compute an approximation to the analytical solution.

3.3.1 Gradient Descent

A popular iterative optimization technique is the gradient descent. This method is based on the observation, that the first derivative converges to zero if the function value approaches the extremum. For example, the minimum of the function $f(x) = \frac{1}{2}(x-4)^2 + 3$ is located at $x = 4$, the derivative $f'(x) = x - 4$ evaluates to $f'(7) = 3$, $f'(2) = -1$ and $f'(5) = 2$, respectively. As clearly visible in this example, the absolute value of f' decreases if the value of x approaches the minimum. The sign of f' states, in what direction (based on the current x) the extremum is located. Figure 3.4 illustrates this principle: The slope of the tangents decreases the closer they are to the minimum.

Gradient descent works by approximating the derivative by computing a delta between two iteration steps. Finding the optimal value of a function f iteratively then consists in computing the next state $x_(k+1)$ (i.e., currently best estimate for the extremum) as follows:

$$x_{k+1} = x_k \pm \gamma \nabla f(x_k), \tag{3.13}$$

where k is current the iteration step, x_0 is an initial guess of the extremum), and $\pm$ stands for either an addition or subtraction, depending on what kind of extremum is searched for: If a minimum should be approached, a subtraction is used, since the goal is to move against the gradient. If a maximum is pursued, the gradient is added. $\nabla f(x_k)$ is the (approximate) gradient at x_k. The factor $\gamma > 0$ is the step size, also called the learning rate. It determines how fast the approximation

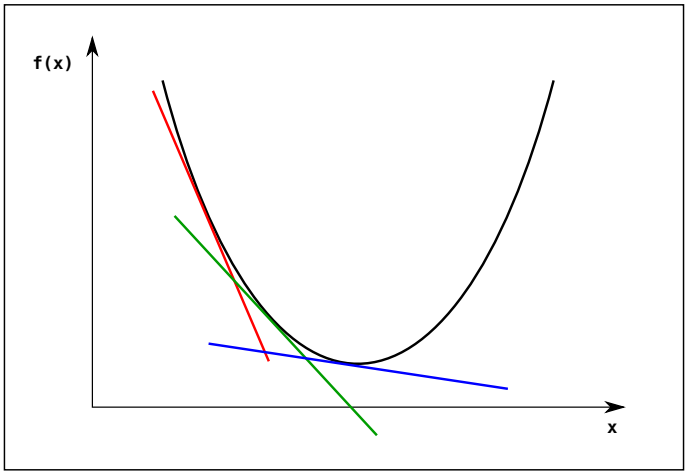

Figure 3.4: Principle of gradient descent.

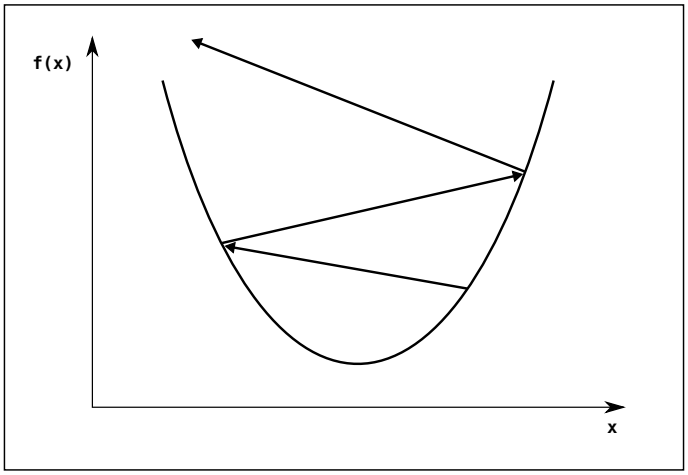

Figure 3.5: Overshooting of gradient descent.

to the extremum happens. A small step size results in a overall slow approximation. However, with a greater the step size a less precise the approximations is possible. The worst case is a divergence of the function f caused by an "escape" in the opposite direction, see Figure 3.5.

Here the step size is too large, which results in a sequence of function values that in turn lead to a greater gradient each step. Therefore it is mandatory to find a trade-off between steps needed to converge and the precision of the approximation. The step size γ could be subject to a dynamic adaption that considers other metrics like the amount of steps computed or even some domain-specific heuristics.

3.3.2 Heuristics

Many problems in computer science suffer from a high computational complexity in order to get the optimal result. Depending on the input size, this could result in very long run times. A heuristic describes a method that allows the computation of a reasonably good result (although not necessarily the optimum), but with a vastly reduced run time. It could be seen as an assessment of the current

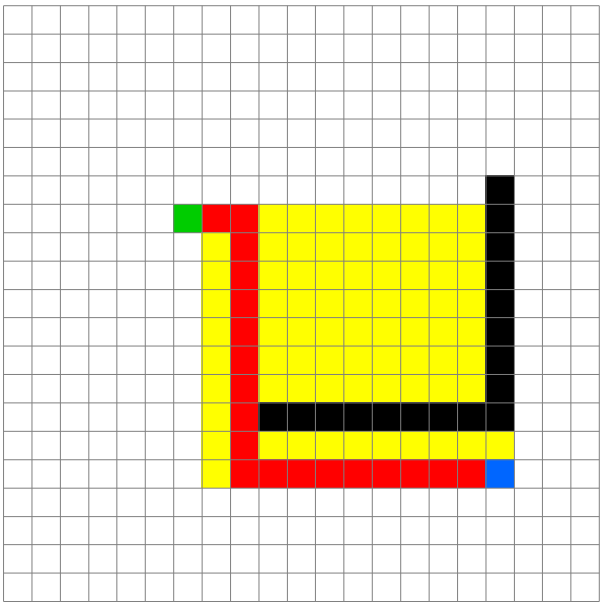

Figure 3.6: Principle of a heuristic: A* search for shortest paths.

state, based on estimates, observations, experience or guesses. This allows for a more informed decision regarding the transition to the next state. In general, heuristics are very domain and problem specific, since they are often tailored to meet special requirements or limitations of the problem. This could be a disadvantage because their specific implementation could not easily be transferred from one problem to another. Besides the run time improvements, a heuristic may be used, too, if the problem at hand does not allow a computation of an exact solution, possibly due to a non-polynomial problem or if not enough information is available.

A popular example is the A* algorithm that solves the shortest path problem. The idea of the heuristic used is shown in Figure 3.6. Here, the search space is explored by computing the linear distance from all the "next-step-candidates" to the target node and choosing the one with the smallest distance. The colors have the following meaning: The agent (green) analyzes nodes (yellow) around itself and in the direction of the target (blue). The path taken is marked red, the obstacle is painted black. Intentionally, the heuristic favors those nodes that are located near to the target, and tries to avoid those further away. It drives the search for the next node of the shortest path in the general direction of the target, i.e., it avoids going in the wrong direction (as a breadth-first search would do). This prevents the excessive exploration of the space in the opposite direction of the target node, and thus reduces the computation time needed when compared to Dijkstra's algorithm.

4 Simulation and Evaluation Concept

By defining an unambiguous system model, misunderstandings regarding the results and claims could be avoided. This chapter introduces the system model used throughout the implementation and evaluation. It consists of three parts that cover the control system itself, its trigger functions, as well as the network abstraction used in the simulation. Next, the problem statement follows in Section 4.2. Based on this model, the proposed concept for the simulation and evaluation is described. It, too, consists of three parts: The offline optimization (which allows a deeper understanding of the NCS simulated), the online optimization with its various trigger parameter adaption policies, and an introduction of the evaluation conducted in Chapter 6.

4.1 System Model

The system model is pictured in Figure 4.1. A set of N different NCS is considered. Each one consists of a sensor (that measures various metrics of the controlled process, called *plant*) and a controller, connected through a shared network.

4.1.1 Control System Model

The time is assumed to be discretized as $t_{k+1} = t_k + T$ and all NCS are sampled synchronously at t_k with the common sampling period T. The control system in each NCS is a discrete time closed loop feedback control system and is modeled as

$$x^i_{k+1} = A^i x^i_k + B^i u^i_k. \tag{4.1}$$

where $x^i_{k+1} \in \mathbb{R}^{n_i}$ is the state and $u^i_k \in \mathbb{R}^{m_i}$ is the input of the system i at time t_k. The following control system matrices A and B correspond to the linearized example used in [LA18] after a discretization of time with a sampling period of 0.01s.

$$A = \begin{bmatrix} 1.0142 & -0.0018 & 0.0651 & -0.0546 \\ -0.0057 & 0.9582 & -0.0001 & 0.0067 \\ 0.0103 & 0.0417 & 0.9363 & 0.0563 \\ 0.0004 & 0.0417 & 0.0129 & 0.9797 \end{bmatrix}, \tag{4.2}$$

$$B = \begin{bmatrix} 0 & -0.0010 \\ 0.0556 & 0 \\ 0.0125 & -0.0304 \\ 0.0125 & -0.0002 \end{bmatrix}. \tag{4.3}$$

The cost function J^i of NCS i used by the Linear Quadratic Regulator is defined as

$$J^i = \sum_{k=0}^{\infty} (x^{i\intercal}_k \, Q \, x^i_k + u^{i\intercal}_k \, R \, u^i_k) \tag{4.4}$$

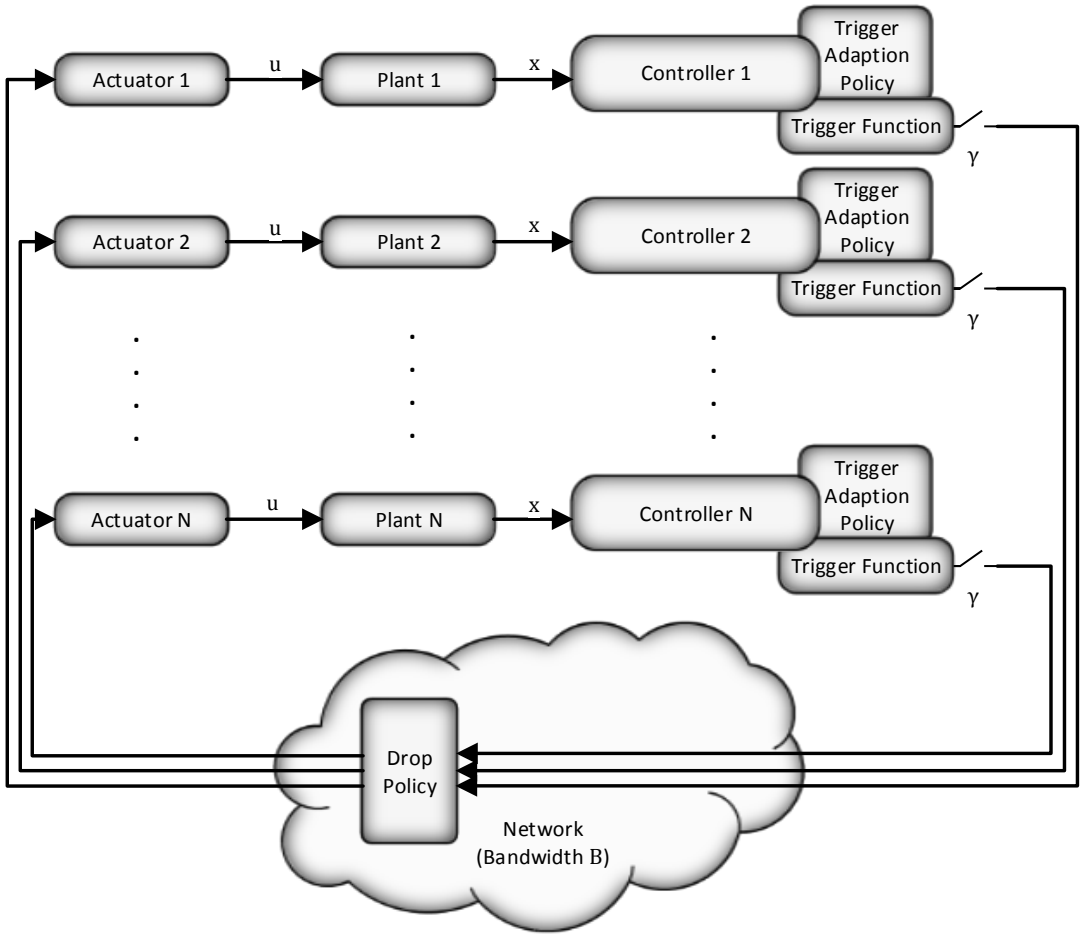

Figure 4.1: System model of the policies described.

The stabilizing controller K itself is an infinite-horizon discrete-time LQR, computed with the discrete-time algebraic Riccati equation as

$$K = \begin{bmatrix} -0.0428633 & -0.9169540 & -0.32993 & -0.820634 \\ 2.4908100 & 0.0750713 & 1.74810 & -1.143350 \end{bmatrix}. \tag{4.5}$$

The penalty matrices Q and R are given as the identity matrix, that is:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.6}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{4.7}$$

The initial state state varies. If the system is not noisy, the state is

$$x_0 = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}^\mathsf{T}. \tag{4.8}$$

If the system is noisy, the initial state is the null vector.

4.1.2 Trigger Functions

Regarding trigger functions, this thesis uses the approach of Linsenmayer and Allgöwer [LA18]. If a sampling occurs at time t_k, a trigger function f_j is evaluated. It determines whether the measured output is transmitted to the controller (and then used as feedback to the compute a new input for the plant) by comparing the trigger function's value with a penalty p_k which is defined as

$$p_k = (u_k - Kx_k)^\intercal (R + B^\intercal PB)(u_k - Kx_k). \tag{4.9}$$

Note that the matrix P comes from the discrete-time algebraic Riccati equation

$$P = Q + A^\intercal PA - A^\intercal PB(R + B^\intercal PB)^{-1} B^\intercal PA. \tag{4.10}$$

The trigger decision γ_k is defined as

$$\gamma_k = \begin{cases} 1 & p > f_j \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

which means that the system triggers (i.e., wants to transmit a packet) if the penalty p exceeds a certain threshold computed by f_j.
The following policies are used in the simulation. All three trigger functions are taken from [LA18].

- Function f_0:

$$f_0(k) = c_0, \quad k \in \mathbb{N}_0 \tag{4.12}$$

  This trigger function resembles the most basic form of triggering: A static threshold c_0 is compared to the current penalty. With $c_0 = 0$, the system triggers every time the function is evaluated. While this would be optimal regarding the performance, the network load is increased drastically. With a larger c_0, the system is allowed to deviate more from the target state, thus the load on the network decreases, but the costs increase.

- Function f_1:

$$f_1(k) = c_1 r_1^k, \quad k \in \mathbb{N}_0 \tag{4.13}$$

  Trigger function f_1 has an explicit dependency on time k associated. With this addition, the trigger frequency increases with the steps simulated since the threshold compared to the penalty decreases (cf. Figure 5 in [LA18]). Note: When simulating noisy systems, a lot of simulation steps are computed. Therefore, a parameter $r_1 \neq 1$ does not make sense, since it would lead quickly to a positive trigger decision every time the function is evaluated, which in turn defeats the overall goal of a trigger parameter adaption. Thus, with noisy systems, the parameter r_1 must be set to 1. This, however, degenerates f_1 to f_0.

- Function f_2:

$$\begin{aligned} &f_2(k, k_\tau, (u_j)_{j\in[\tau_{k_\tau}, k-1]}, (u_j^c)_{j\in[\tau_{k_\tau}, k-1]}) \\ =&c_2 r_2^{k_\tau} - \sum_{j=\tau_{k_\tau}}^{k-1} (u_j - Kx_j)^\intercal (R + B^\intercal PB)(u_j - Kx_j) \\ =&c_2 r_2^{k_\tau} - \sum_{j=\tau_{k_\tau}}^{k-1} p_j, \\ &k \in 0 \cup [\tau_{k_\tau} + 1, \tau_{k_\tau+1}], k_\tau \in \mathbb{N}_0 \end{aligned} \tag{4.14}$$

The authors of [LA18] define the sequence τ as $\tau := (\tau_{k_\tau})_{k_\tau \in \mathbb{N}_0}$ with $\tau_0 := 0$ and $(\tau_{k_\tau})_{k_\tau \in \mathbb{N}_0}$. In contrast to f_1, this trigger function holds an internal state between two evaluations. It is dependent on time in two ways: Directly through the the parameter k (that affects the sum of penalties) and implicitly by k_τ (i.e., the time instants of previous, successful trigger decisions). The same reasoning as with f_1 regarding noisy systems and the parameter $r_2 = 1$ applies, although it does not degenerate to f_0.

4.1.3 Network Abstraction

As shown in Figure 4.1, the network slice connecting each "smart sensor" group (i.e., a plant and its control system) and the actuator is shared among all N NCS but no other applications (i.e., no cross-traffic).

It has a static bandwidth ("link capacity") B, which allows at most B packet transmissions at time instant t_k. If the amount of packets scheduled at time t_k exceeds B, excess packets are dropped according to the drop policy δ. The relative link capacity B_r (in percent) describes the ratio of the available bandwidth B to the number of NCS N. Given that all NCS trigger at one time instant, a relative link capacity of $B_r = 15$ % means that at most 15 % of the packets are transmitted over the network. The two policies used are *random dropping* and *static priority dropping*, which are explained in detail in Section 6.1.3.

One packet contains all input information used in one control cycle, i.e., no splitting of information across multiple packets occurs.

The order of events concerning packets and the network is as follows: At time instant t_k, for every NCS i, the current status of the plant $x^i_{t_k}$ is measured. Then, a trigger decision $\gamma^i_{t_k}$ is made, which could result in a packet sent over the shared network slice. Before it possibly reaches the actuator, excess packets are dropped, so that the link capacity requirement B is satisfied. Note that the time instant t_k is the same for all NCS, i.e., the amount of packets considered by the drop policy δ equals to $\sum_{i=0}^{N} \gamma^i_{t_k}$.

4.2 Problem Statement

At every sampling instant t_k, noise is added to each vector component of the state x^i_k of the control system described in Section 4.1.1. The noise level is determined by the normal distribution $\mathcal{N}(\mu, \sigma^2)$ with the mean $\mu = 0$ and variance $\sigma^2 = 1$. It is scaled by a noise factor $\phi \in \mathbb{R}_{\geq 0}$.

With multiple NCS sharing a network with bandwidth limits B, the problem investigated in this thesis is how the trigger parameters of trigger functions f_1 and f_2 could be adjusted to obtain the best control system performance.

The parameters of the trigger functions, in particular c_j, describe how often a system tries to send a packet over the network. Given the scenario that every NCS triggers at every time instant t_k. This would lead to a congestion which results in a lot of packet drops. In the other case, the NCS trigger too infrequently. While this reduces the network load, the control system's performance is drastically reduced since too little information is fed back to the input.

Intuitively, a trade-off exists between triggering too often and too rarely. One possible solution is to adapt the trigger parameter c_j in order to drive the control system into that optimum. This could be done by means of a trigger adaption policy. Four proposed approaches are described in Section 4.3.3.

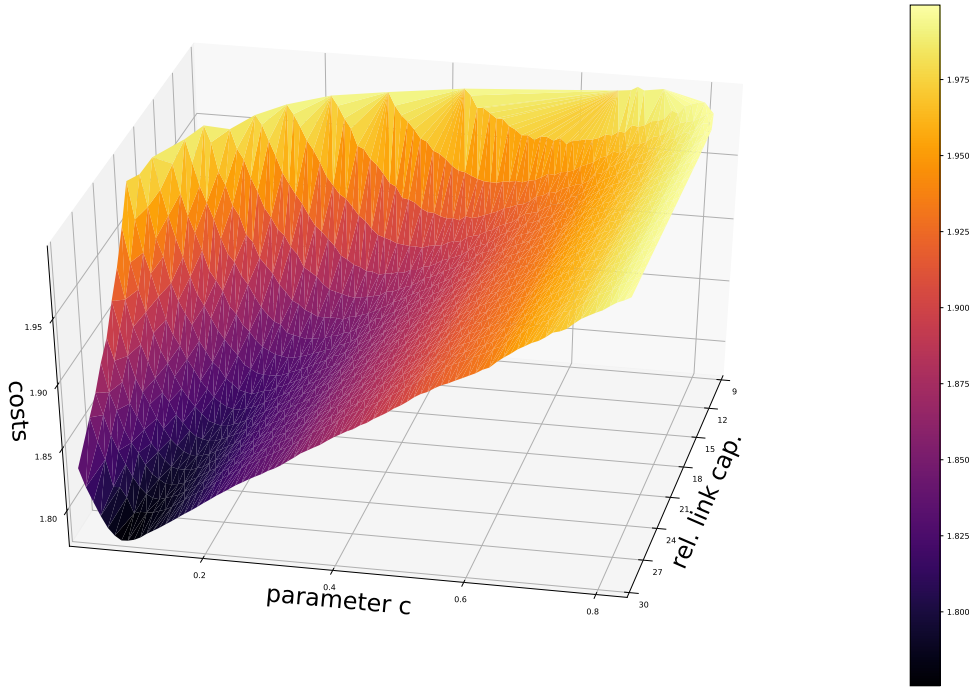

Figure 4.2: Offline Optimization: Exhaustive search of parameter space. Note the "valley" of minima.

4.3 Proposed Concept

4.3.1 Offline Optimization (Exhaustive Search)

In order to get a better grasp of the various parameters influencing the optimal value of the parameter c_j regarding the control system performance, an exhaustive search for the optimum is conducted. Since a LQR is used as controller, the cost function has the characteristics of a quadratic function. It is convex and has well defined minimum, which is not known in advance. This minimum of costs depends on the link capacity B, the noise factor ϕ and the trigger function f_j. The other influences from the control system itself are omitted in this study because only one control system is used. In general, though, the matrices A, B, Q and R do have an influence.

The exhaustive search is done by simulating many scenarios using a parameter combination of the parameter space (i.e., the cross-product $\mathcal{S} = B \times \phi \times c_j$). Since $|\mathcal{S}|$ increases exponentially with the amount of values a single parameter can assume, their range and resolution must be chosen carefully. The exhaustive search is therefore limited in its depth, i.e., precision of the results. With too many possible values, the simulation count grows too quickly and the time required makes a complete search infeasible. The values obtained by the offline optimization are used to compare the performance of the trigger adaption policies to an optimum. They enable the iterative development and improvement of the policies covered in Section 4.3.3.

Figure 4.2 shows the result of the offline optimization for f_1 and $\phi = 0.08$. The main characteristics of such a scenario are visible:

- The convex nature of the function is verified.

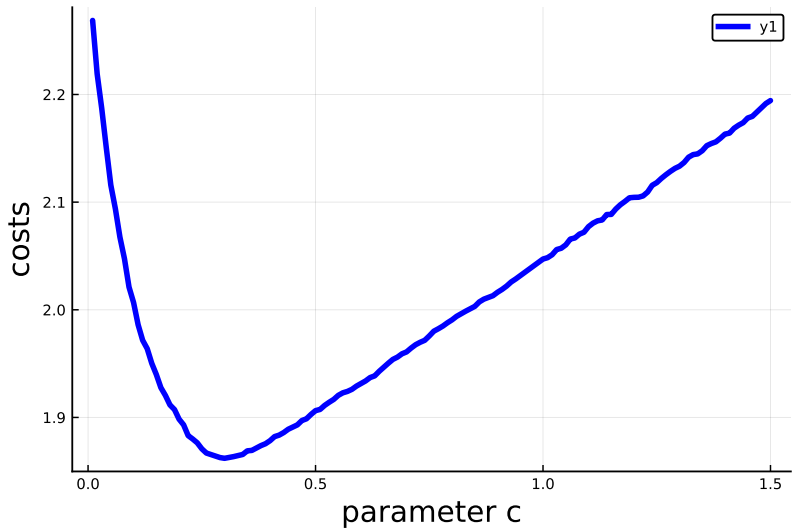

Figure 4.3: Example: Costs relative to parameter c. Relative link capacity: 15 %

- Although the parameter space for c_1 is $[0, \infty)$, the optimum corresponds to a value close to zero ($0 < c_1 < 0.5$).
  Compare this with the choice of $c_1 = c_2 = 20$ in [LA18]. However, the system investigated in this paper is not noisy, hence a parameter $r_i \neq 1$ is possible.
- The optimal value of c_1 depends greatly on the relative link capacity B_r (see also Section 6.1.2).
- The decrease in performance diminishes with a larger link capacity B_r.
- With a parameter c_1 close to zero, the costs soar very high.

Figure 4.3 shows a selected example that illustrates the points above in more detail. It is a slice of the surface with the link capacity being $B_r = 15\%$.

4.3.2 Online Optimization (Parameter Adaption)

The previously described exhaustive search serves as a basis for the actual contribution of this master thesis. Instead of setting the parameter c_j once at the beginning, it should be adapted dynamically, i.e., to possibly changing environments. This is achieved by an additional step to the computation of γ_{t_k}. Just before the trigger function is evaluated, the parameter adaption policy can change the trigger parameter $c_{j,k}$, which now becomes time-dependent, too.

The parameter adaption policy uses various metrics of the system in question to compute a step size, which is then added to $c_{i,k}$. Given the convex characteristic of the cost function, the optimization technique proposed is the gradient descent explained in Section 3.3.1. Essentially, it determines a direction and magnitude to modify the optimized parameter in order to minimize the associated cost function. Other optimization techniques such as simulated annealing could also be used, but are disregarded in this contribution because the function optimized does not have multiple minima.

The following characteristics of the overall scenario must be considered when developing an adaption policy:

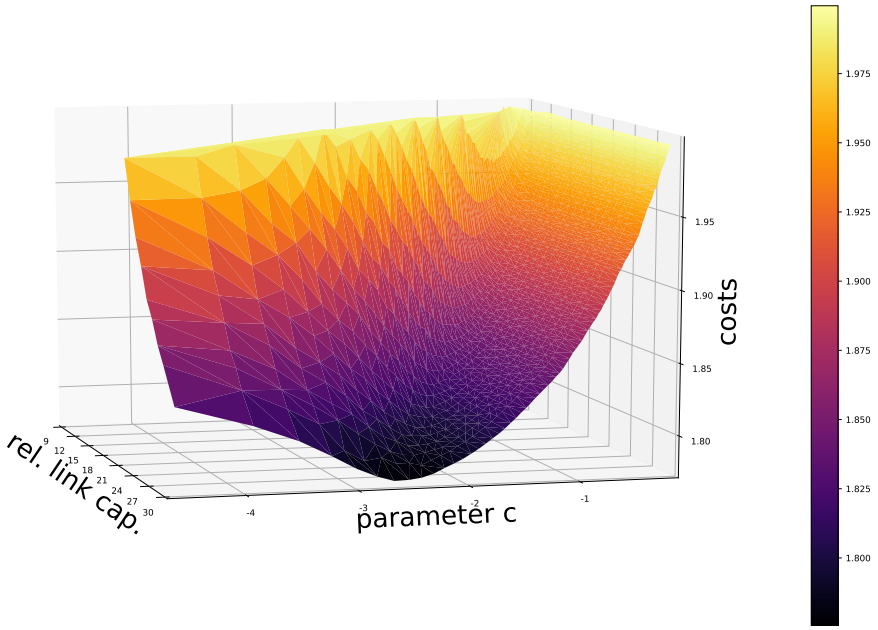

Figure 4.4: Transformation of parameter c into logarithmic space.

Noise

Since the system is noisy, the gradient computed is noisy, too. This leads to a reduced precision since the actual gradient must be estimated. The uncertainty is compensated by averaging the costs of multiple previous time steps using a static "window size".

Behavior of costs with c_j close to zero:

Figure 4.3 shows the non-linear behavior of the costs with the parameter c_j left to the optimum (i.e., $c_j \leq c_{j_{opt}}$). Here, the costs growth is much larger that right to the optimum, which leads to a higher sensibility of the gradient towards small changes (often induced through noise). The influence of the dissimilarity of the left and right side of the optimum on the gradient could be reduced by transforming the parameter c_j (for which the gradient is computed) into a logarithmic space, as seen in Figure 4.4. The transformation is done by computing the natural logarithm, i.e., $c_{j,log} = ln(c_j)$. To compute the step size in the original domain, the gradient computed in logarithmic space must be transformed back from the codomain by the inverse function, i.e., e^x.

Overshooting:

If the parameter c_j becomes negative, $\gamma_k = 1$ at the next sampling instant. Since this leads to an increase of packets sent (and therefore dropped), the probability that next packet packet is dropped, too, rises. In effect, this is a self-perpetuating process, which essentially deadlocks the trigger adaption policy. It is therefore mandatory to keep the parameter c_j non-negative. This could be easily achieved by using the log-space transformation described above, since the natural logarithm stays positive for every input value.

Trigger Rate, Drop Rate

The trigger rate r_t describes, how often a NCS triggers (i.e., it wants to transmit a packet) in relation to all opportunities it could have triggered. The same holds for the drop rate r_d: It states, how often a packet is dropped in relation to all the time instants it was possible to drop a packet, with $r_d \leq r_t$.

These two metrics could be used in a heuristic (refer to Section 3.3.2) that improves the overall performance of a NCS. The heuristic is based on the observation that both r_t and r_d, as well as their ratio, take on a specific value if the trigger policy operates at its optimum. Obviously, these values depend on all the parameters the optimum depends on, namely the available link capacity c, the noise factor ϕ, and the trigger function (e.g., the parameter c for trigger function $f1$). A possible approach is to increase the trigger parameter c, as long as the r_d/r_t-ratio is not at its optimal value. Since the overall nature of the problem places the optimal trigger parameter c very close to zero, the heuristic could be used in conjunction with another "force" in a way that the parameter c is both pushed away from zero by the heuristic's influence (which diminishes farther away from zero) and pushed to zero by the gradient.

Figure 4.5 shows the trigger and drop rates of three simulations with a static (i.e., not adjusted) trigger parameter c. The first plot illustrates the rates for a c well below the optimum, the middle one displays the same scenario with the optimal c value, and the rightmost picture shows a simulation with a c greater than the optimum.

Centralized vs. Decentralized Adaption

When multiple NCS are simulated, the adjustment to the trigger function's parameters could be made either "en bloc", i.e., simultaneously for all systems, or for every single NCS separately. The advantage of the former is that information about every NCS are available to compute a globally optimal reaction to the systems.

While this makes the computation of averages and therefore the gradient less prone to errors induced by disturbances that affect only a single system, this is also its biggest weakness. In the case that the disturbance affects multiple systems and is of such nature that it could be handled (i.e., controlled) optimally at the affected systems only, the centralized approach falls short: It would adjust parameters at all NCS, regardless if they are impacted by said disturbance or not. This could hurt the overall performance significantly. The other approach is to control and adjust all systems individually. Depending on the setting, this could improve the performance or deteriorate it further. Figure 6.14 shows a scenario of the decentralized approach which yields degraded performance compared to the centralized method shown in Figure 6.10.

Regarding performance, the obvious issue is that some individual systems deviate far from the near-optimal other systems. To some extend, this behavior could be improved by increasing the window size, as described in Section 6.1.6. This, though, leads to an increased stabilization and reaction (to an external disturbance) time. In this thesis, both approaches are evaluated, since the best fit for an application depends greatly on the limitations and requirements the environment imposes on the NCS compound: While in one scenario, a quick and individual reaction to external events is favored at the expense of some performance, other settings require the absolute best performance possible and can exclude major disturbances.

4.3.3 Proposed Adaption Policies

Based on the characteristics described above, the following four trigger parameter adaption policies are proposed and investigated.

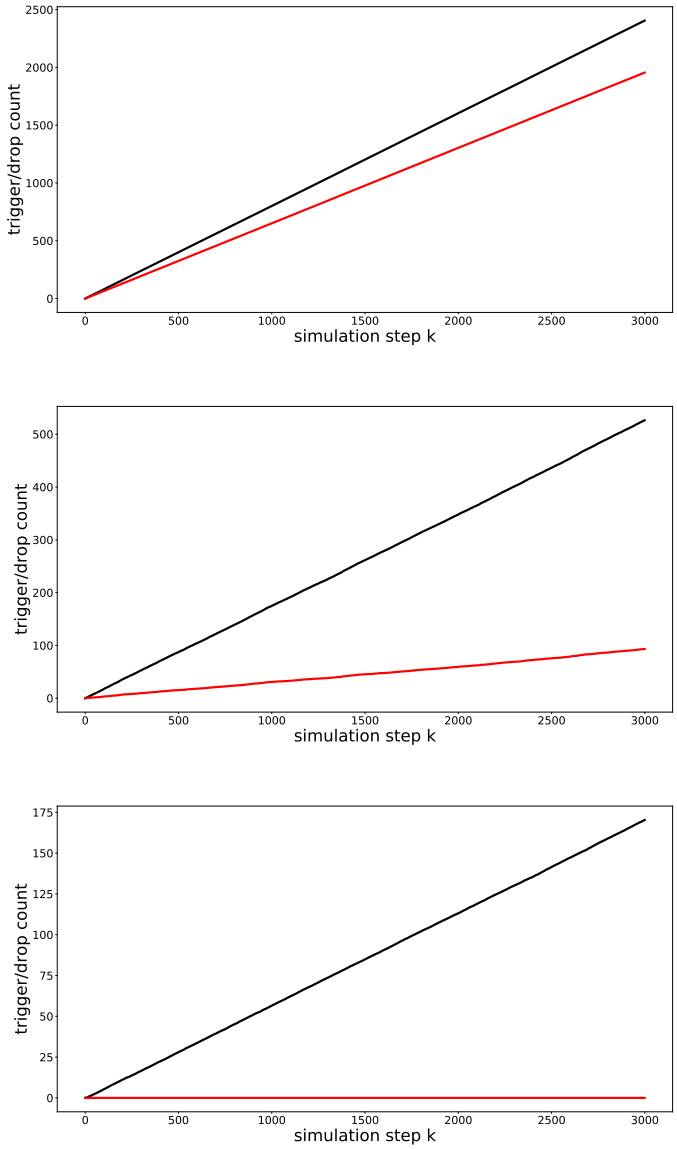

Figure 4.5: Trigger and drop rate influenced by parameter c.

Policy 1: Centralized Adaption The main feature is the centralized nature of this approach. This means that the parameter c_j of all NCS is adapted to the same value by the computation of its gradient (considering the average of the costs of all NCS). c_j is modified as long as the gradient is not zero. The resulting step size is capped by a static limit (see Section 6.1.7 for a discussion), which avoids very large steps. In addition, overshooting into negative values is prevented.

Policy 2: Decentralized Adaption This approach is the decentralized version of policy 1. The parameter c_j is adapted for each NCS individually. The state held in the trigger function is local to the NCS Like the centralized counterpart, the step size is capped. Overshooting is prevented, too.

Policy 3: Centralized Adaption in Log-Space This policy is an enhanced version of policy 1. Again, the parameter c_j is adapted for all trigger functions simultaneously by using a metric's average of all NCS. The gradient is now computed in the log-space described in Section 4.3.2.

Furthermore, before computing a new step size, the gradient is modified. The idea behind this modification is to trust the gradient more if the previous step size (and thus the gradient) was large. This decreases the influence of noise, since the gradient is relatively small if parameter c_j approaches the optimum (but then the relative influence of noise increases). The transformation to and from log-space ensures that c_j cannot become negative.

Additionally, the trigger and drop rate are measured and used in a heuristic corresponding to the one described in Section 4.3.2.

The method used to adapt both the gradient (with the gradient correction) and step size (using the heuristic composed of the trigger and drop rate) is as follows: For example, the influence of the gradient is larger if it is trusted more, which in turn depends on the previous step size σ. This is implemented by computing a gradient correction factor ω, which is defined as

$$\omega = \kappa \cdot \eta^{-|\sigma|}, \tag{4.15}$$

where κ is a constant in the range $[0, 1)$ describing the maximum gradient correction factor and η is given as $\eta = 2^{\frac{1}{\upsilon}}$. The term υ describes at what step size the correction factor should be $\frac{1}{2}\eta$. The same kind of diminishing influence (given by $constant^{-|value|}$) is used with the heuristic (see 6).

One can observe that the gradient alone causes a parameter c_j that is a little too small. This is compensated by "pushing" it away from zero with the correction from the drop rate heuristic. In total, two forces need to be parameterized: The one corresponding to the gradient (pushes to the left, towards zero) and the one corresponding to the trigger and drop rate heuristic (pushes to the right, away from zero).

The maximum step size is capped in the same manner as with policy 1.

Policy 4: Decentralized Adaption in Log-Space The fourth policy has the same characteristics as policy 3. The only and major difference is that adjustments to c_j are made to each system individually, which requires NCS-specific storage space to hold the state needed for the gradient and the heuristic.

4.3.4 Evaluation

Aggregation of Results

When simulating a noiseless system for k steps, the performance of the system is given by the accumulated costs at the kth step, see Section 6.1.1.

When simulating a noisy system, the performance is evaluated by computing the mean costs of the last w steps. Furthermore, the costs are normalized with respect to all K NCS. To minimize the influence of a specific random sequence produced by the pseudo-random number generator (PRNG) and reinforce the validity of this contribution, different seed values are used. In total, the costs are then averaged considering K systems, the last w steps of each system's performance, and every seed value the simulation has been run with.

Comparison of Results

For the comparison of results, several combinations of system parameters are possible, including the trigger function f_j, the trigger parameter adaption policy (policies 1 through 4), the initial step size used in the adaption policy, the window size used in the adaption policy, the drop policy and the link capacity.

To allow a quick assessment of the trigger adaption policy in question, every approach is compared to the optimum (see Section 4.3.1. Furthermore, an alternative approach called “naive approach” is evaluated.

In this policy, the trigger decision γ is redefined such that $\gamma^i_{t_k} = 1$. This means that every NCS triggers at every time instant t_k, regardless of the system state, the penalty or the trigger function f_j.

5 Implementation

This chapter covers the development of the software. The term "software" usually describes all artifacts created during the development [LL13]. It is used in this chapter mostly in a more narrow sense, though, and describes in particular the source code as well as the executable binary. First, the requirements imposed by the proposed concept of Chapter 4 are described. Next, desirable properties of the architecture are stated. Then, the architecture of the software is presented in Section 5.2.1. After technical details of the Implementation are described, extensions to the system are covered and possible future improvements pointed out.

5.1 Requirements

The main goal of the software developed is to address the problem statement of Section 4.2. Thus, the software should be able to

simulate control systems The question investigated revolves around the performance of control systems, with and without noise. The software should therefore be able to simulate such systems for a specific amount of steps.

simulate multiple NCS The system model, in particular the network abstraction, covered in Section 4.1 requires that multiple NCS using a shared network with a limited bandwidth could be simulated.

support multiple drop policies The main focus is on random dropping. Nevertheless, the system should be able to simulate multiple drop policies.

provide NCS performance In order to compare different trigger adaption policies, their performance must be compared. This requires an extraction of at least the performance metric for both noisy systems and those without noise.

support multiple trigger functions A trigger function is necessary in order to adapt any trigger parameters. Multiple trigger functions are needed to allow an easy comparison of different approaches.

support multiple trigger adaption policies The contribution of this thesis is the development and comparison of different performance optimizing adaption policies. Therefore, the ability to simulate different adaption policies is absolutely necessary.

support seed values Seed values are used for the sequence of numbers a PRNG produces, which in turn are used for the noise generation as well as the random dropping policy. To allow statistically valid claim concerning the performance results, multiple seed values must be supported.

aggregate results As described in Section 4.3.4, the results of multiple NCS must be aggregated in order to allow a reasonable comparison of different parameter adaption approaches.

configure scenarios To allow an easy adjustment of simulation parameters, scenarios consisting of multiple simulations should be configurable. In particular, this means that it should be easy to specify a range for an individual parameter.

5.2 Software Architecture

Based on the requirements stated in the previous section, a software architecture is developed. This architecture should adhere to the following goals, i.e., non-functional requirements.

Scalability A software system is scalable if it can handle workloads of growing size. Considering the given requirements, the software should be able to simulate a lot of scenarios without reaching the limits of the host system, such as memory boundaries.

Maintainability Software aging is the process of erosion of a given software system. This erosion comes in the shape of multiple smaller or larger changes to the system, that affect the costs associated with future modifications. A well-designed architecture minimizes these costs and could reduce the amount of subtle bugs introduced by future adjustments. Besides the architecture, a plethora of methods and guidelines exist that improve or preserve the maintainability, most of which do not apply to the architecture. These approaches often try to prevent so-called "Code Smells". See [KPG09] for an analysis of code smell impacts, their root causes and methods of mitigation.

Extensibility Software systems are often extended to increase their performance or add new features. If this fact is considered while developing the architecture, future extensions to the software could be made with minimal effort and costs. This is especially important since multiple trigger functions, drop and parameter adaption policies should be supported. By being extensible, it is easy to add such functionality to the software in the future.

Distributability If a software satisfies this property, it means that the software or parts thereof could be distributed across multiple systems or environments. This term is closely related to the scalability. With this property, a software could potentially handle much larger workloads because by being provisioned to more or more powerful machines.

Usability Usability describes how easy it is to use a software. For many systems, the usability is of crucial importance for its acceptance among users. Note that usability does not automatically mean that a Graphical User Interface (GUI) is required. It rather states that needed functionality should be easy to access. If a system is highly usable it does not imply that it is simple to do so.

5.2.1 Architecture of the Software System

The functional and non-functional requirements lead to the architecture shown in Figure 5.1.

There are seven modules of the simulator in total. Each corresponds to one or more requirements and fulfills a specific task. The interaction of following two modules is of particular importance:

Simulation The main hub of the software is this simulation module. It executes the very core of the software, i.e., it simulates a control system by computing subsequent states. To fulfill this task, it requires information about the drop policy, trigger function and the trigger adaption policy,

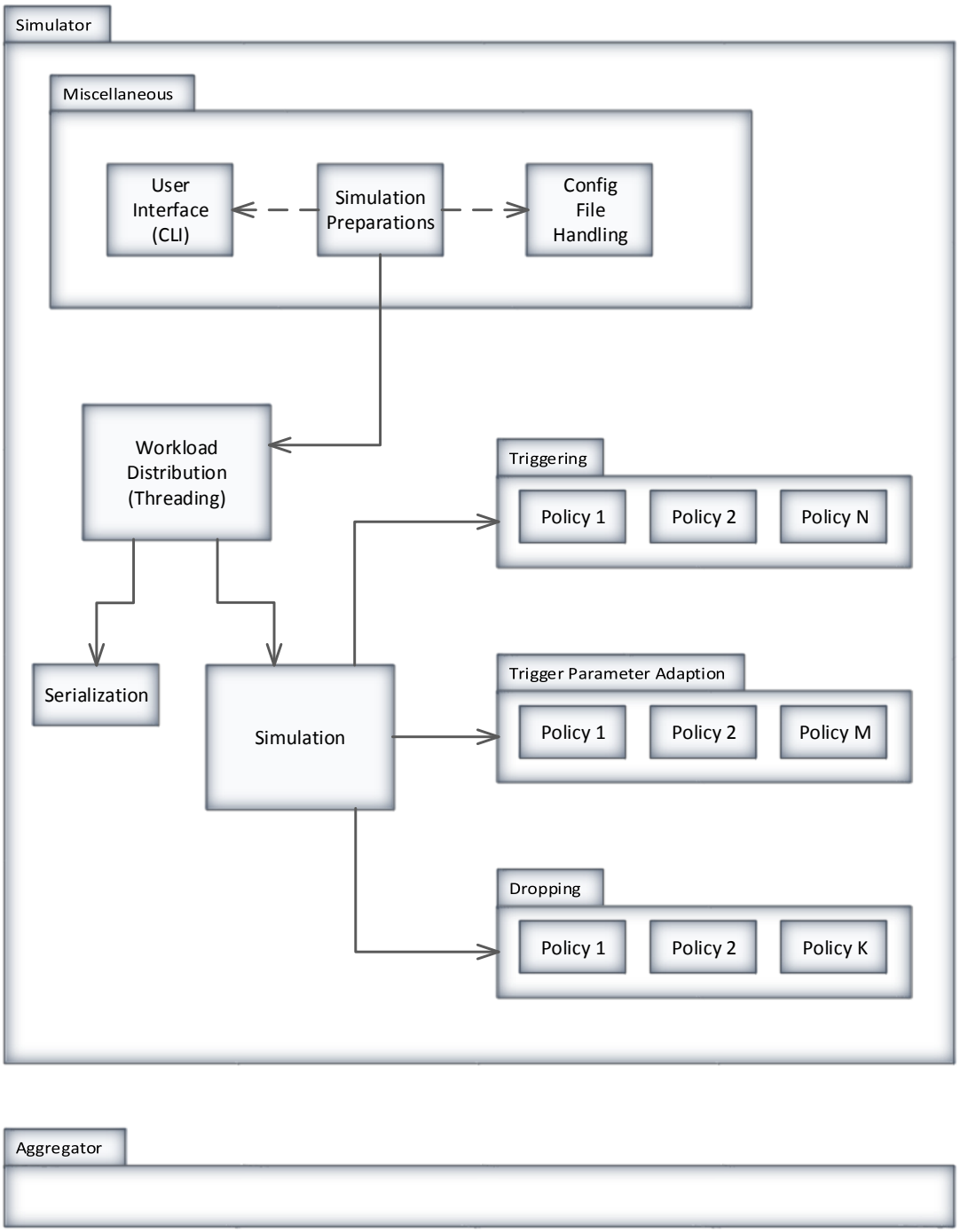

Figure 5.1: Software architecture of the implementation.

and is therefore dependent on these modules. In this context, the term *simulation* means the execution of k steps, i.e., simulating the behavior of of the respective control system, trigger function, trigger parameter adaption policy and network with its drop policy.

Workload Distribution This module serves as runtime governor and schedules the overall processing. This is done in multiple steps. First, a configuration file is read and the parameters from the command line are parsed. Second, based on this information, a list of abstract packages is compiled. These packages called "RunConfigs" contain all information necessary to execute a single simulation. This includes data structures, their allocated memory and the actual parameters of each policy used. By handling these isolated packages, no state is held between two simulations. Third, the list of RunConfigs is split among available worker threads. The property of state isolation in particular makes it possible to support the simulation of hundreds of thousands simulations. Additionally, this module handles the continuous serialization of the results, which reduces memory requirements drastically.

The Aggregator module is an isolated subproject. It is independent of the other modules, both on a code basis and at runtime. The aggregator processes the output from the simulator, i.e., the raw performance output of every NCS simulated.

From the software engineering point of view, modularity is a very important property. It has a strong influence on the modifiability (i.e., how easy it is to modify the software) of the code and is closely related to the "separation of concerns" principle. Following this principle means to map individual elements of the problem to individual elements of the solution. This has the effect that modifications of one component concern this very component only (in contrast to concerning other, logically unrelated components, too). A commonly used mean to achieve modularity are (software) interfaces. These manage the access to the functionality provided by the module and simultaneously encapsulate implementation details. Good modularization is associated with a lose coupling of the modules and a high individual cohesion of each module. The employed architecture satisfies these requirements and thus yields a high maintainability and extensibility.

5.3 Implementation Details

5.3.1 Workflow

When using this software, the typical workflow is as follows:

1. Write a configuration file. It contains all information needed to execute several simulations. In particular, it

2. Simulate. By using the configuration file as input to the simulator, all possible parameter combinations are compiled and the corresponding simulations are then executed.

3. Aggregate the results. Using the aggregator to aggregate the results of different simulations.

4. Analyze the results. To understand individual simulations, it is possible to extract the trajectories of some metrics like the costs or amount of packets dropped so far. Visualizing these values by plotting them often helps with the troubleshooting.

5.3.2 Used Technologies and Dependencies

In the development of the software, multiple tools are used, which are presented in the following paragraphs. The software is designed to run locally on one individual host. It does not distribute its execution state across several machines, although such a concept could be possible (see Section 5.4).

IDE

To write the source code itself, the Integrated Development Environment (IDE) CLion[1] is used. It simplifies the software development process by providing functionalities like auto-completion, build system and compiler integration or tools for refactoring code.

Version Control

A version control allows a precise change management including a history. The source code management used is Git[2]. It supports a distributed and potentially decentralized development process. Its maturity and prevalence allow an easy integration on nearly all development systems.

Programming Language

The software itself is written in the C++ and uses features of C++11 like smart pointers. This very mature, object oriented programming language is supported by a multitude of platforms and compilers.

Build System

In order to support different development environments, the build system CMake[3] is used. Essentially, a build system is another level of indirection in the build process. It abstracts from the actual underlying system (i.e., run-time environment), compiler and operating system. Furthermore, it allows a time efficient build process by rebuilding only those files that have changed since the last builds, thus reducing the coding-test-cycle drastically. CMake is easily used in conjunction with CLion. The compiler used is the GNU C++ Compiler from the GNU Compiler Collection (GCC)[4].

Used Libraries

The software uses the following libraries:

- Eigen[5], a very large library covering almost all aspects of linear algebra. Used for the simulation of control systems.
- Clara[6], a library used to parse the command line options.

[1] https://www.jetbrains.com/clion/
[2] https://git-scm.com/
[3] https://cmake.org/
[4] https://www.gnu.org/software/gcc/
[5] http://eigen.tuxfamily.org/
[6] https://github.com/catchorg/Clara

- json11[7], a parser used to read JSON files.
- StopWatch[8], a library for measuring time.

5.3.3 Simulator

Configuration Files

The simulator executes a scenario that is described in a configuration file. This has several advantages like reproducibility and automatic documentation of parameters used. The format used for this task is JSON. JSON[9] is a data serialization and interchange format. Its hierarchical structure as well as human-readability fit the requirements of configuration. Listing 5.1 shows a sample configuration file with only one class, i.e., a homogeneous scenario. Parameter ranges could be given in the `(start:increment:end)` syntax.

Runtime Behavior

After parsing a configuration file, the RunConfig queue is created. It contains all independent simulations that need to be executed for one scenario. By marking specific RunConfigs prior to their execution, it is possible to extract information from the simulation. This allows a further investigation of the time-dependent behavior of various metrics like the trigger rate or performance.

This queue is then accessed by multiple worker threads. Each worker thread executes a RunConfig, saves its results and fetches the next RunConfig from the queue. This parallelization allows an optimal utilization of available resources like multiple CPU cores.

Since the queue is a shared data structure, access to it must be sequentialized. This is done by protecting the queue with a mutex. In order to reduce the locking overhead, a worker thread takes multiple RunConfigs at once (a block) and manages an internal, thread-local queue.

RunConfig are independent of each other and could be computed in any sequence by any amount of threads. To avoid interference between threads, thread-local data structures must be used in order to simulate a RunConfig. This concerns in particular the PRNG used for the noise and random dropping, the adapted parameter of a trigger function, trigger function state, and so forth.

Various techniques are used to accelerate the execution of a simulation, e.g., when simulating centralized trigger adaption policies. Since the parameter adjustment (step size) is the same for all N NCS, it is computed only once and saved in every step instead of computing it N times for each NCS individually.

Because the computed results could sum up to tremendous amounts of data, a worker thread saves its results in a persistence queue, which is then accessed by a special persistence thread, that writes the data every i seconds to disk. This access is mutex-protected, too.

The results are written in CSV format. This allows an easy handling among a multitude of tools, including spread sheet calculations and scientific programming languages like Matlab or Julia.

[7] https://github.com/dropbox/json11
[8] https://github.com/KjellKod/StopWatch
[9] https://www.json.org/

Listing 5.1 Example: Configuration file in JSON format.

```
{
    "scenario_name" : "classes-test",
    "scenario_comment" : "foobar comment",
    "seed_values" : "[(1:1:1)]",
    "drop_policies" : [1, 2],
    "link_capacity_percentages" : "[(15:1:15)]",
    "steps" : 30000,
    "extraction_count" : 1,
    "cost_average_window_size" : 1500,
    "cost_delta_epsilon" : 0.2,

    "is_noisy" : true,
    "noise_factors" : "[0.08]",

    "sensor_classes" : [
        {
        "class_name" : "c1",
        "sensor_count" : 20,

        "control_system" : 1,
        "cs_sensor_initial_state" : "[0.0, 0.0, 0.0, 0.0]",
        "cs_stabilization_time_steps" : 200,

        "trigger_function" : 2,
        "trigger_param_1" : "[10]",
        "trigger_param_2" : "[1]",

        "trigger_adjustment_policy" : 4,
        "trigger_adjustment_factor_1" : "[0.5]",
        "trigger_adjustment_factor_2" : "[1000]",

        "sensor_priorities" : "1, 2, 3, 4, 5, 6, 7, 8, 9"
        }
    ]
}
```

Listing 5.2 Parent class TriggerFunction

```
class TriggerFunction {
public:
    bool get_trigger_decision(const Eigen::MatrixXd& regulator_feedback,
        const Eigen::MatrixXd& regulator_state,
        std::vector<double>& trigger_state,
        double param_1,
        double param_2) {
            double penalty = get_penalty(regulator_feedback, regulator_state);

            bool result = trigger_decision_nvi(p, trigger_state, param_1, param_2);
            return result;
        }

    double get_penalty(const Eigen::MatrixXd& regulator_feedback,
        const Eigen::MatrixXd& regulator_state) const;

private:
    virtual bool trigger_decision_nvi(const double penalty,
        std::vector<double>& trigger_state,
        double param_1,
        double param_2) const = 0;
};
```

Selected Code Example: Trigger function

Listing 5.2 shows the essential inner workings of the `TriggerFunction` class. Conceptually, the penalty p is computed in the same manner, independent of the actual trigger function. Since multiple trigger functions should be supported, a combination of polymorphism and the non-virtual interface pattern (see Scott Meyer's Item 35 in [Mey05]) is used. Note that the public interface hides the inner workings of both the parent class and the subclasses. The latter could influence the result, though, by implementing the private and pure virtual method `trigger_decision_nvi()`.

Listing 5.3 displays an excerpt of trigger function 1. Note that the penalty is computed and compared to the actual value that f_1 computes in the parent class. The child class in turn cannot change this behavior. It could only implement `trigger_decision_nvi()`.

The simulation itself then accesses the actual trigger result through a smart pointer of type `TriggerFunction`. Dereferencing it dispatches the function call `get_trigger_decision()` at runtime internally to one of the child classes.

5.3.4 Aggregator

The aggregator processes a CSV by parsing it and combining the entries. This is done according to the split-apply-combine paradigm. For more information on this subject, see [Wic+11].

Listing 5.3 Child class TriggerFunctionF1

```
class TriggerFunctionF1 : public TriggerFunction {
private:
bool trigger_decision_nvi(const double penalty,
    std::vector<double>& trigger_state,
    double param_1,
    double param_2) const override {
        auto is_triggered = (penalty > (param_1 * pow(param_2, trigger_state[0])));
        trigger_state[0] += 1.0;
        return is_triggered;
    }
};


```

5.4 Possible Improvements

Although the software is sufficient fast, it could be improved with the following modifications:

- Distribution across multiple hosts: Using multiple threads has speed limitations imposed, for example, by the CPU architecture or bandwidth of memory buses. By distributing the scenario over multiple hosts, this disadvantage could be mitigated. A commonly used and probably fitting framework is OpenMP[10].
- Dedicated matrix multiplications: The core operation of the simulator is the multiplication of matrices. Regarding computation speed, general purpose processors are very slow compared to dedicated chipsets like those used in Graphic Processing Units (GPU). Leveraging their speed advantage could reduce the time required for one simulation significantly.

[10] https://www.openmp.org/

6 Evaluation

This chapter covers the qualitative and quantitative analysis of the results obtained with the implementation. It complements the concept presented in Chapter 4. Due to the complex nature of the subject and the large amount of combinatorial possibilities of the different parameters belonging to a simulation, a quick overview over the most important parameters and their influences on the control system's performance is given. After this qualitative discussion, a quantitative analysis is conducted by comparing the performance of the trigger adjustment policies introduced in Chapter 4 with the optimum (i.e., no dropping at all, transmit feedback at every time step) and the naive solution (i.e., with dropping, but with feedback transmission at every time instant).

6.1 Qualitative Discussion, Selected Examples

In this section, the qualitative aspects of the adjustable parameters of a system are addressed. By making a comparison of the optimal behavior with the following illustrations and short descriptions, it is possible to get get a better grasp of the individual influence of the specific parameters. Note that the time is discretized, which allows a transmission of data only at specific time instants that are the time period T apart.

6.1.1 Optimum

The optimal trajectory of the control system's performance depends neither on the available link capacity, nor on any trigger parameter or its adjustment behavior. In fact, the optimum (i.e., minimal) costs are achieved by feeding back the output of the control system at every time instant. Furthermore, it is necessary that no packet is dropped, so at every time instant, the link capacity is sufficient to allow every packet sent to be received.

Figure 6.1 shows a typical trajectory of a single control system. The y-axis shows the accumulated costs of the system, instead of the "stage costs", i.e., the costs that occur additionally at each time step. Although the system model assumes multiple NCS sharing the same network, the trajectories are the same since the individual control systems are (from an analytical point of view) indistinguishable.

After the system has stabilized, accumulated costs of approximately 354 are reached.

When simulating noisy systems, the average costs of the last few steps is considered (instead of the accumulated costs with noiseless systems). Furthermore, the performance of all NCS is aggregated by computing the mean value. Figures 4.2 and 6.2 illustrate the influence of the link capacity on the performance achievable in noisy systems. The optimum values for noisy systems come from the exhaustive search described in 4.3.1. See Section 6.1.4 for the (in comparison to the link capacity relatively small) influence of the noise factor ϕ. Note the usage of a the random drop policy.

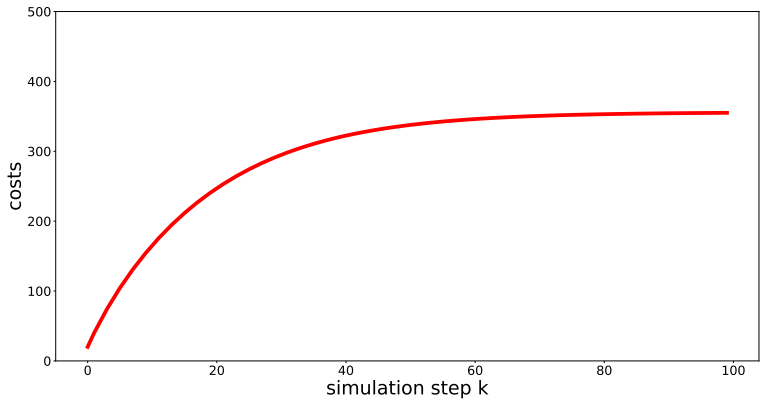

Figure 6.1: Optimal trajectory of a NCS.

6.1.2 Link Capacities

The link capacity states, how many packets could be transmitted at a specific point in time over the network. Note: As introduced in Section 4.1.3, this is a simplification of the actual, in reality happening process. Nevertheless, it still allows a reasonable analysis and easy implementation.

If the amount of packets that is scheduled to be transmitted exceeds the available link capacity, packets are dropped. There are various policies that describe how to decide which packet to drop, with two of them being described in the next Subsection. To illustrate the influence of the link capacity on the performance of a single system, multiple simulations are run and grouped according to the aggregation technique described in Section 4.3.4.

The drop policy used in this scenario is “random dropping”, which means that at specific time instant, packets are selected and discarded randomly until the link capacity is large enough to let all packets pass through. Since it is very difficult to gather true entropy in order to get real random numbers (which in turn are required to make the drop decision), a PRNG is used (for more detail on this subject, see Section 4.3.4, too). The sequence of pseudo-random numbers (and therefore the respective packet dropped) depends on the seed value used. It is therefore mandatory to average the results over multiple seed values in order to get a statistically sound statement on the simulation results.

Figure 6.2 shows the overall performance (y-axis) of various link capacities (x-axis). Note that the y-axis has been shifted to the best achievable value, which is 3 (refer to 6.1.1).

As it can clearly be seen, the performance is worse the lower the link capacity. Intuitively, this makes sense, since the probability packet is dropped is inversely proportional to the link capacity. With more packets dropped, the performance suffers because the output of the control system is not fed back and therefore could not be used to compute a new input u.

6.1.3 Drop Policies

Given the scenario that at a specific point in time, of the 100 NCS do $s = 50$ control systems want to transmit a packet. The link capacity shall be $c = 40$ packets, which means that the amount of packets scheduled for transmission exceeds the available network link capacity. Complying with the system model, this leads to a reduction of scheduled packets to c, i.e., $d = s - c = 10$ packets must be dropped.

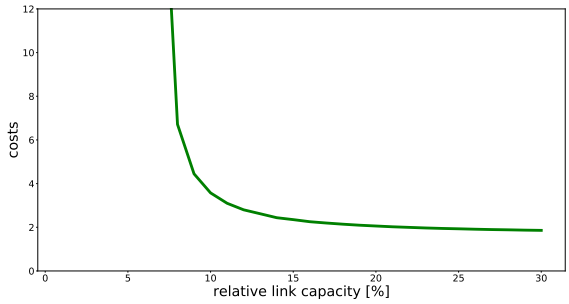

Figure 6.2: Influence of different network link capacities on the control system's performance. Noise factor $\phi = 0.08$.

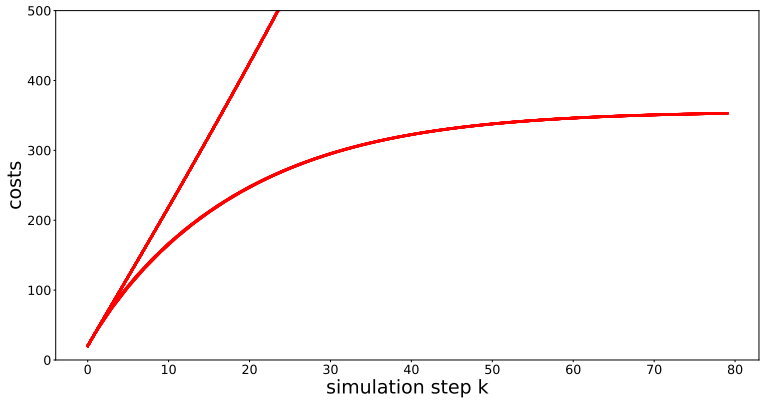

Figure 6.3: Static priority dropping favors some control systems only. Their performance does not suffer, while those systems with a lower priority have diverging costs.

A drop policy is a specification, how these $d = 10$ packets are selected among all scheduled s packets. Two policies have been implemented:

Random Dropping With this approach, d packets are selected randomly among the s packets that should be transmitted.

Static Priority Dropping In contrast to random dropping, every NCS gets a static priority assigned. When choosing the packets to be dropped, those coming from a system with a lower priority are picked preferably, i.e., a lower priority leads to less packets transmitted and therefore to higher costs.

This relation is clearly shown in Figure 6.3. While some systems with a higher priority stabilize, since their packets are transmitted, those with a lower priority have diverging costs.

6.1.4 Noise and Noise Amplification Factor

Some control systems need to deal with system internal uncertainty. This uncertainty may come in many shapes such as

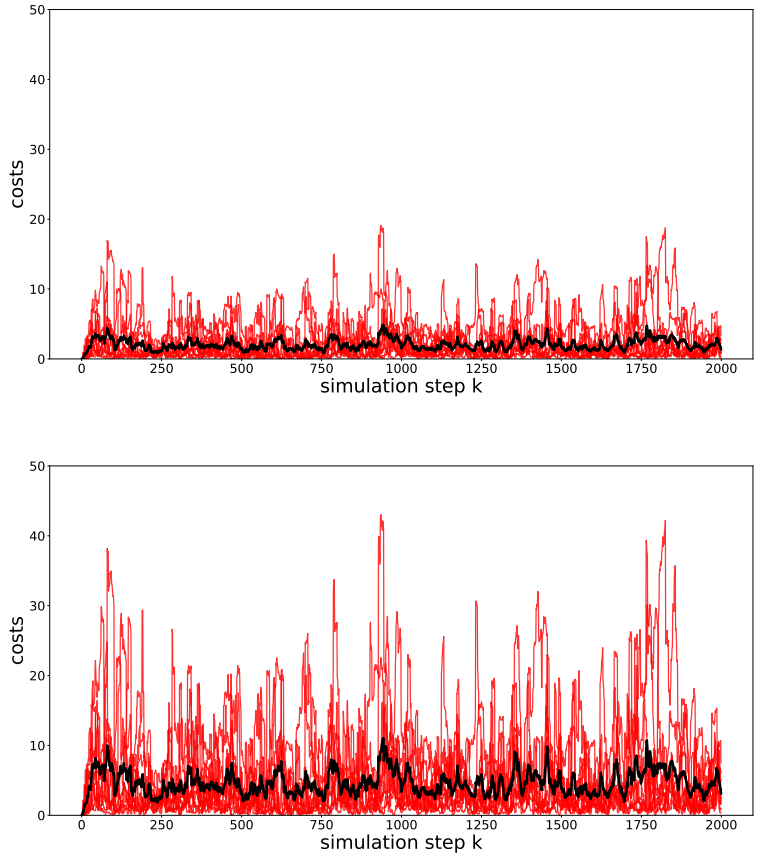

Figure 6.4: Different noise factors ϕ affect the system's performance.

- modeling errors, i.e., the control system does not meet the assumed specifications exactly.
- noisy measurements, i.e., the value measured by a sensor is not (exactly) the real value of the physical process.
- noisy actuation, e.g., an actuator could not be controlled as precisely as the input dictates it.
- noisy physical process, i.e., the underlying process is (partly) random in nature, e.g., radioactive decay

By adding noise to the simulation, the behavior of such noisy systems could be modeled. This behavior, in particular the performance, of a system may vary greatly depending on the amount of noise added. A "noise factor" is used to control this property of a simulation. Figure 6.4 displays the same set of NCS but with two different noise factors $\phi = 0.08$s and $\phi = 0.12$. The seed value used in the PRNG is in all three simulations the same. Clearly visible is relation between noise and performance. A different noise factor does influence the magnitude of the performance, but it does not other the behavior. This applies obviously only, if the system triggers every at every sampling instant (i.e., $\gamma := 1$). Since the penalty considers the system performance, the behavior would indeed change with a different trigger function.

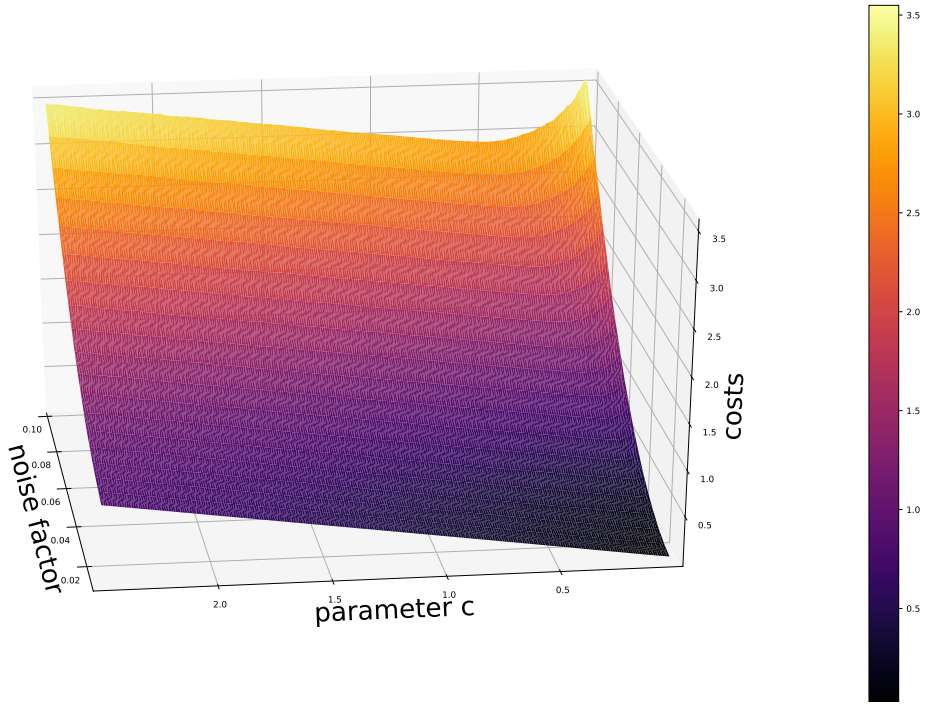

Figure 6.5: Performance of a NCS in relation to the noise factor ϕ and the trigger function parameter c. Note the varying optimum location that depends on ϕ.

Considering the trigger function $f1$ from 4.1.2, a more subtle relationship exists between the location of the minimum with respect to the parameter c and the noise factor. This is illustrated in Figure 6.5. The surface plot shows the performance of the system, the parameter c of the trigger function and different noise factors ϕ. The general tendency is: The more noise is added to a system, the larger the parameter c in the cost minimum. Note that with a higher trigger threshold c, another effect comes into play: If the system does not trigger often, the performance suffers because the system could benefit from the feedback loop.

6.1.5 Number of Individual Control Systems

The settings investigated in this master thesis involve many individual NCS. Here, the total count of NCS simulated is of crucial importance. If few systems are used, one NCS influences the result far more than if a lot of systems are used. This is mainly due to the type of aggregation used to combine the result of each NCS into a single number. While the median would not suffer from one outlier, the average does. In contrast, the median would not be the best choice when using drop policies like priority dropping: If more than half of the NCS are able to transmit, the median would indicate no issue, although a lot of systems may have divergent costs. In this kind of setting, the susceptibility of the average to outliers is indeed advantageous since a single failing or otherwise badly performing NCS is reflected directly in the computed average. This avoids overlooking a potential problem and fits the requirements of real-world NCS better.

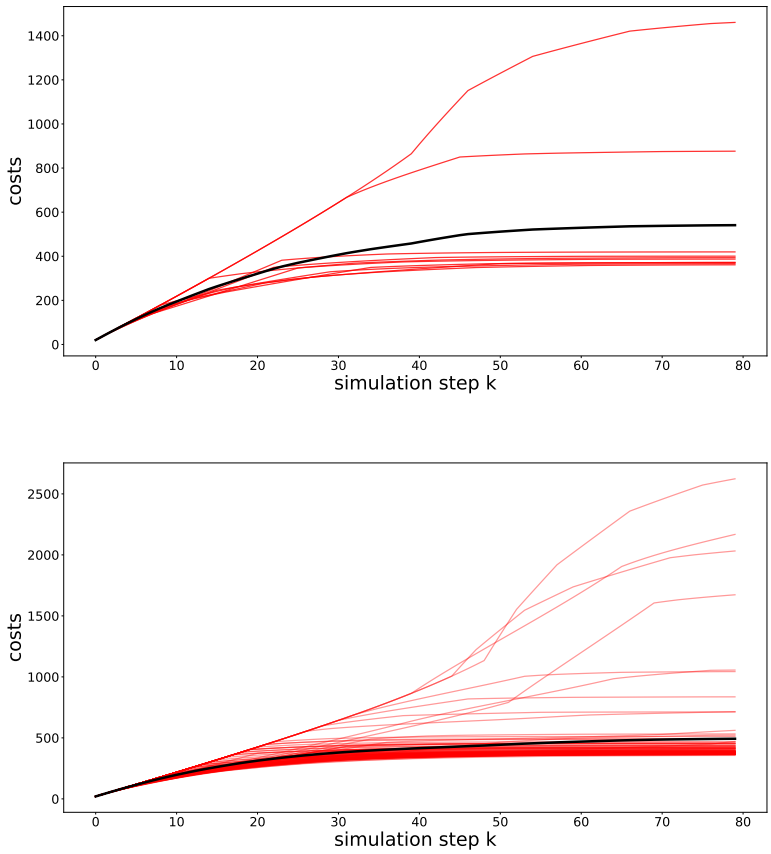

Figure 6.6: Influence of the NCS count used in the simulation on the performance.

In order to use the average as aggregation, preferably a lot of NCS are simulated. Figure 6.6 shows the same scenario as above, once simulated with 10, once with 100 NCS. In the 10 NCS scenario (upper plot), the average costs are 540. However, the impact of a few outliers is reduced with more systems, although their magnitude increases. The average costs of the 100 NCS scenario (lower plot) are just 492. Note that the average is plotted in black.

6.1.6 Window Sizes

In this simulation setting, the window size w determines, how many values are considered to compute the average of a metric. For example, it is used when averaging the costs of the last w steps in order to compute the difference to the previous window's average (which in turn is used to compute the gradient). Scaling this parameter properly could improve the performance of a NCS by finding the sweet spot between a lower value (which leads to a shorter reaction time, but makes the average more susceptible to outliers) and a higher value (which increases the precision of gradient computation, but leads to a slower reaction time). Note that the size of the window greatly depends on the dynamics of the control system.

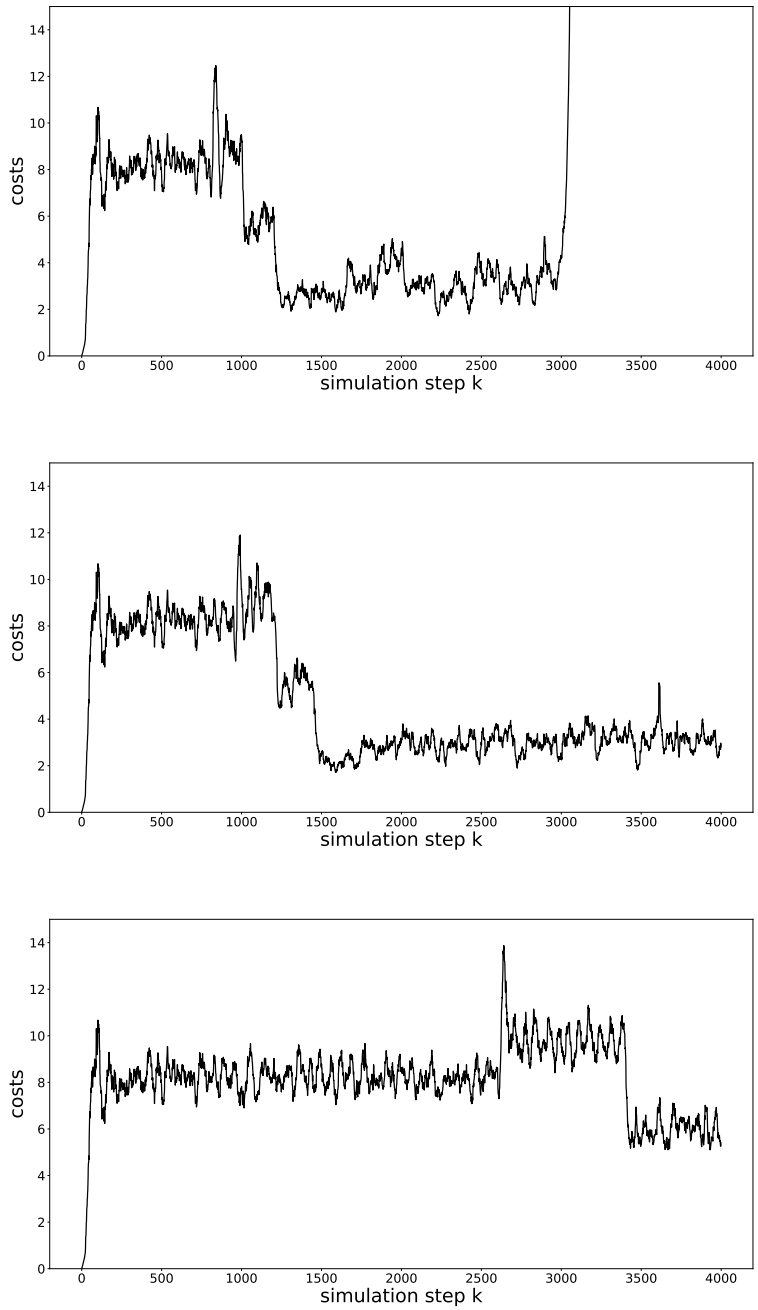

Figure 6.7: Influence of the window size on the reaction speed and the performance.

Figure 6.7 illustrates these two extremes: The upper image shows a small window of 200 steps, the middle one a reasonable choice of 500 steps and the bottom image a window size of 800 which could be considered too large in this scenario. It is clearly visible how the small window size could lead to a divergence of performance due to an incorrect gradient computation (upper plot). The plot shows the mean value of all NCS at simulation step k.

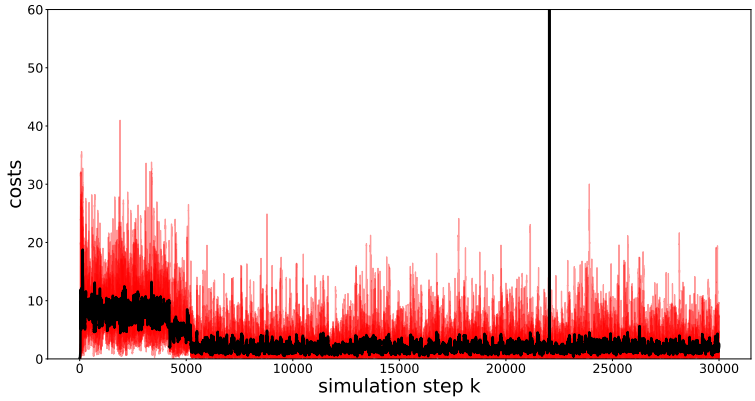

Figure 6.8: Sudden cost peaks after long simulation runs.

6.1.7 Simulation Steps

The amount of steps equals to the time the control system's state is allowed to evolve. If the simulation is run for only very few steps, various effects may not be visible. These effects include in particular, but are not limited to, the stabilization of the system and a seemingly strange divergence of the parameter c of trigger function $f1$. In Figure 6.8 it is conspicuous that the peak of costs occurs after a very long time (in relation to the stabilization time at the beginning). Often, this peak happens at a single NCS only. Note that the trigger adjustment policy 3 is used. Besides the difficulty of detecting such peaks (it is unclear if such a peak would occur again after more simulation steps), their mitigation is equally intricate. One approach is to limit the step size. As the quantitative analysis below shows, this doesn't solve the problem completely, in particular with decentralized adaption policies.

The main reason for these peaks is the gradient computation of the trigger adjustment policy: If the system is nearly perfectly stable, very small changes in the state induced by the noise lead to a relatively large gradient, which in turn influences the parameter c step size (i.e., the adjustment relative to the previous value) directly. Over the course of very few adjustments, this escalates to a very large step size and thus to a very large parameter c, which in turn leads to extremely high costs, since a packet is virtually never sent anymore. The trigger adaption policies proposed (see Section 4.3.3) try to minimize this effect by limiting the step size.

Furthermore, the step count is tightly related to the window size w described earlier, since at least w steps must be simulated, before an average of the last w steps could be computed.

6.2 Quantitative Analysis, Comparison of Adaption Policies

The benchmark metric is the overall performance J, averaged over all NCS and seed values. The performance of the trigger adaption policies (introduced in Section 4.3.3) is evaluated in the following paragraphs. In order to allow an easy comparison to the optimum and naive approach, a semi-transparent "corridor" consisting of these two reference trajectories is included in the plots. The figures show the performance with respect to the relative link capacity B_r. Each policy is described

				simulation steps		trig. adp. window	
seed values	**noise factor**	**GD LR**	**initial trig. param.** c_j	**Policy 1+3**	**Policy 2+4**	**Policy 1+3**	**Policy 2+4**
100	0.08	0.5	20	30,000	100,000	1,000	10,000

Table 6.1: Simulation parameters used in the evaluation.

by two plots. The first displays the absolute costs, the second shows the relative performance (relative to the optimum and naive approach). In every plot, the approach discussed in shown with a thick blue line. The optimum in turn is plotted in thin green, the naive approach in thin red.

Note: In general, due to the higher probability of a NCS getting its packet dropped multiple times in a row, the performance significance regarding low relative link capacities is very small. In particular, this concerns bandwidths $B_r \leq 5\ \%$.

If not stated otherwise, the simulation parameters for all examples discussed are given in Table 6.1. Note that the window size changes from centralized to decentralized policies. The term "GD LR" means gradient descent learning rate and describes the default step size that is modified by the gradient and heuristic.

6.2.1 Policy 1: Centralized Adjustment

The following figures correspond to the most simple centralized approach:

- Used trigger function: f_1, absolute costs: Figure 6.9.
- Used trigger function: f_1, relative costs: Figure 6.10.
- Used trigger function: f_2, absolute costs: Figure 6.11.
- Used trigger function: f_2, relative costs: Figure 6.12.

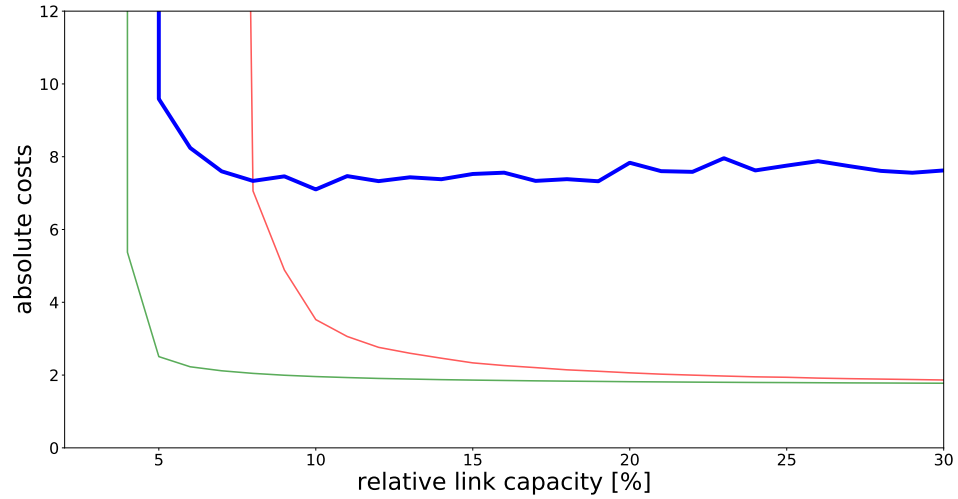

Figure 6.9: Policy 1, trigger function f_1, absolute costs

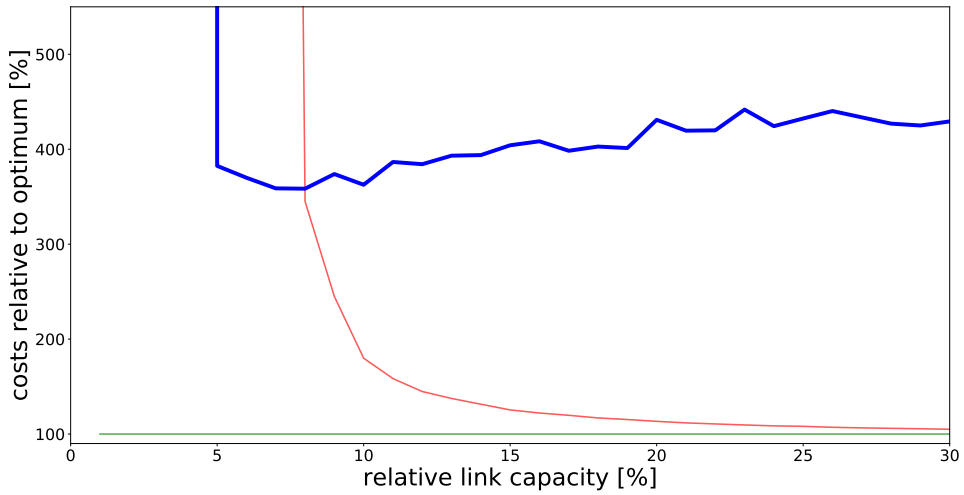

Figure 6.10: Policy 1, trigger function f_1, relative costs

The performance with trigger function f_i is better only in the small range of about $5\ \% \leq B_r \leq 7\ \%$. If the other trigger function f_2 is used, this range could be extended up to $14\ \%$. With a link capacity above this threshold, the costs relative to the naive approach increase drastically for f_1 and less severe for f_2. This approach could therefore be considered as unsuitable for real-world applications.

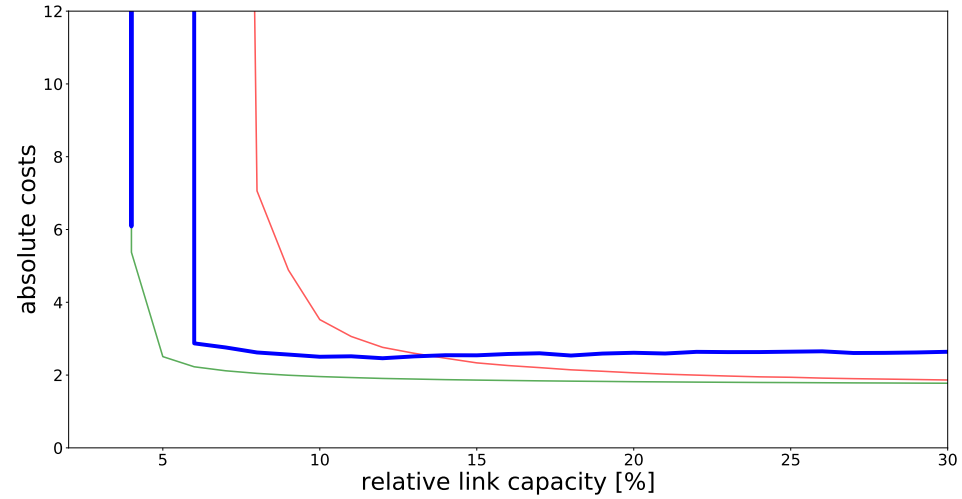

Figure 6.11: Policy 1, trigger function f_2, absolute costs

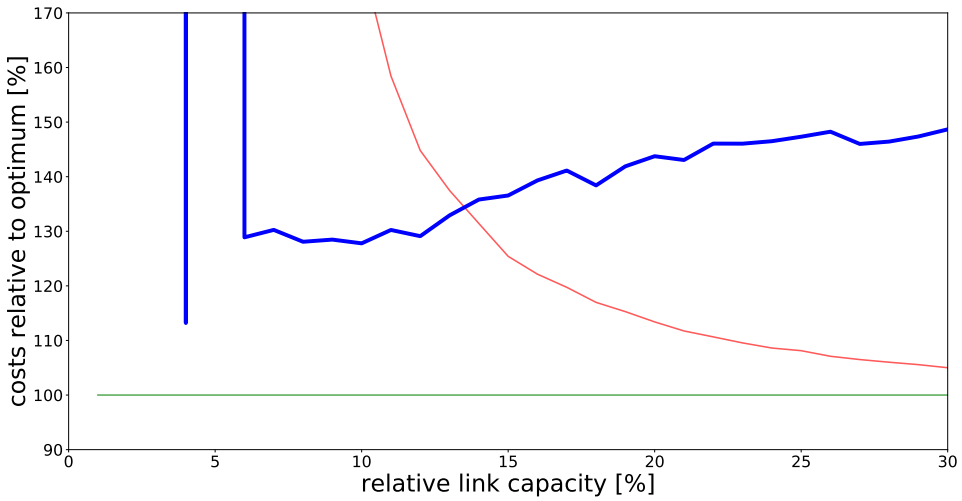

Figure 6.12: Policy 1, trigger function f_2, relative costs

6.2.2 Policy 2: Decentralized Adjustment

The following figures correspond to the most simple decentralized approach:

- Used trigger function: f_1, absolute costs: Figure 6.13.
- Used trigger function: f_1, relative costs: Figure 6.14.
- Used trigger function: f_2, absolute costs: Figure 6.15.
- Used trigger function: f_2, relative costs: Figure 6.16.

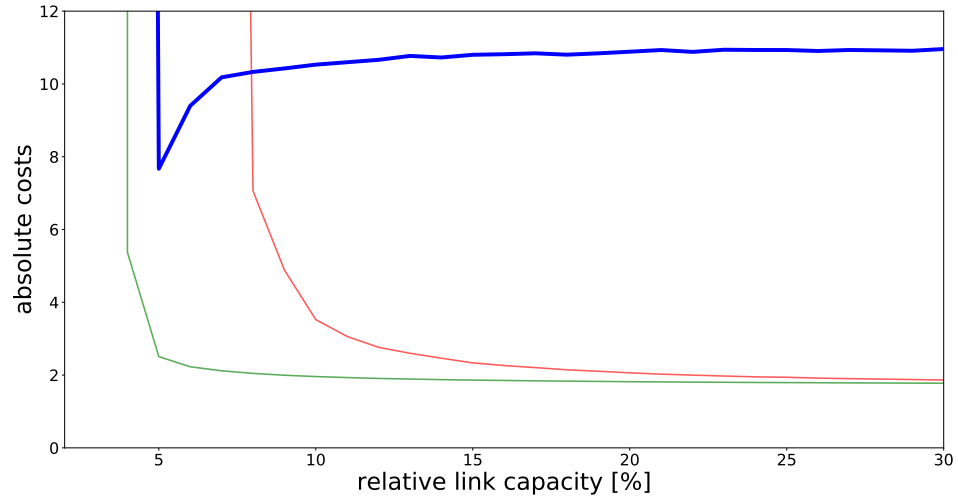

Figure 6.13: Policy 2, trigger function f_1, absolute costs

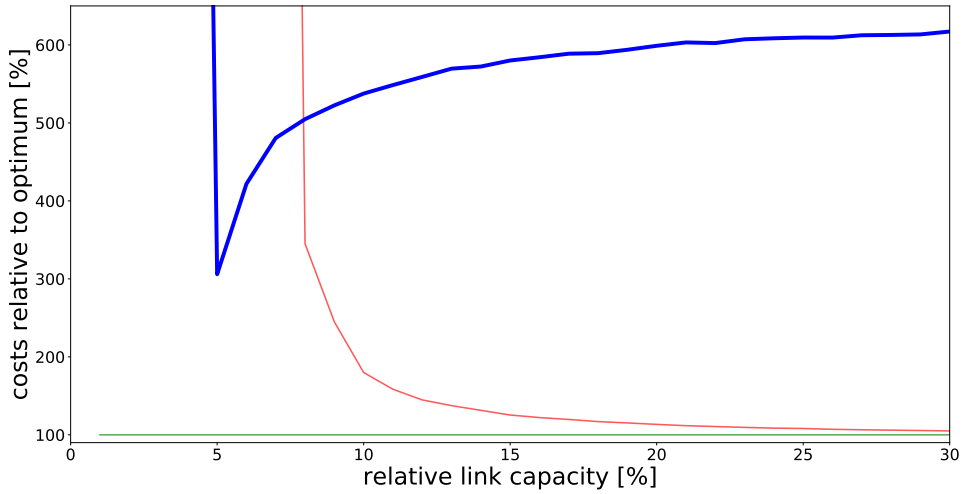

Figure 6.14: Policy 2, trigger function f_1, relative costs

The statement given for policy 1 applies almost exactly for this approach. Note that the performance is slightly worse compared to policy 1, possibly due to the effects described in Section 4.3.2.

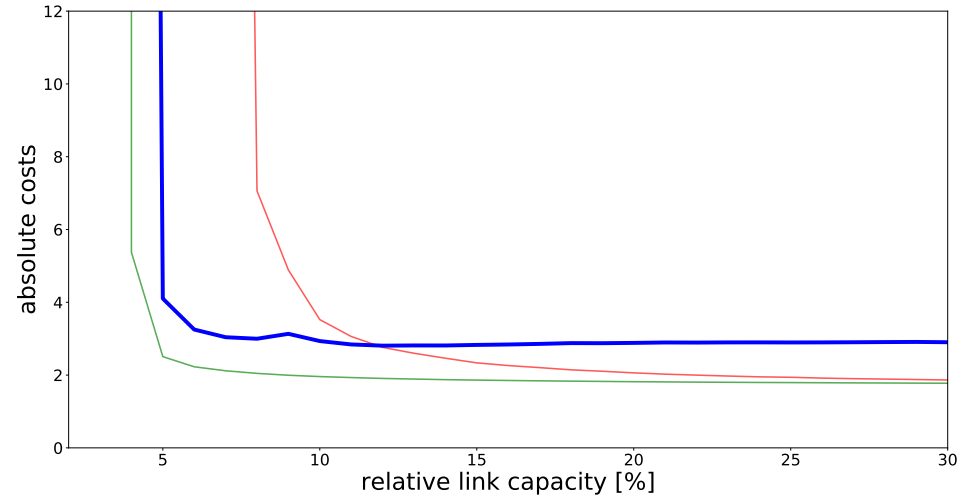

Figure 6.15: Policy 2, trigger function f_2, absolute costs

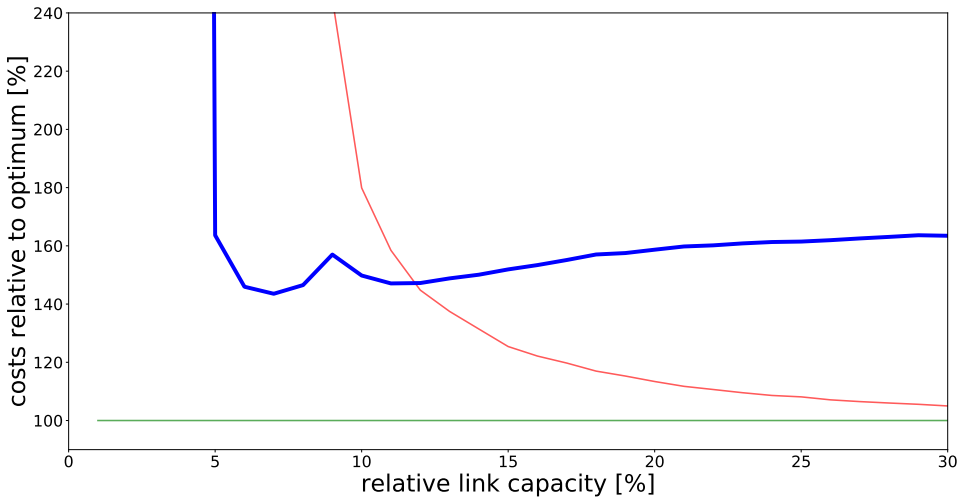

Figure 6.16: Policy 2, trigger function f_2, relative costs

6.2.3 Policy 3: Centralized Adjustment in Log-Space

The following figures correspond to the more sophisticated centralized approach:

- Used trigger function: f_1, absolute costs: Figure 6.17.
- Used trigger function: f_1, relative costs: Figure 6.18.
- Used trigger function: f_2, absolute costs: Figure 6.19.
- Used trigger function: f_2, relative costs: Figure 6.20.

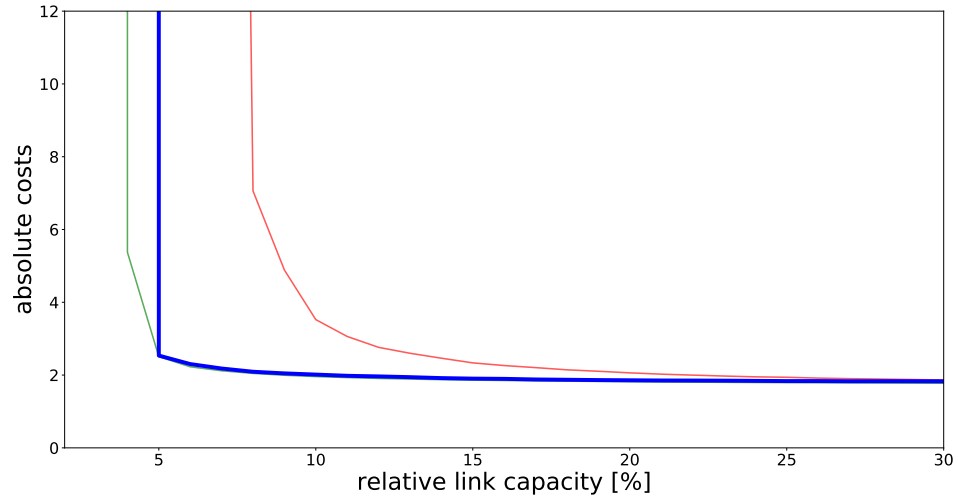

Figure 6.17: Policy 3, trigger function f_1, absolute costs

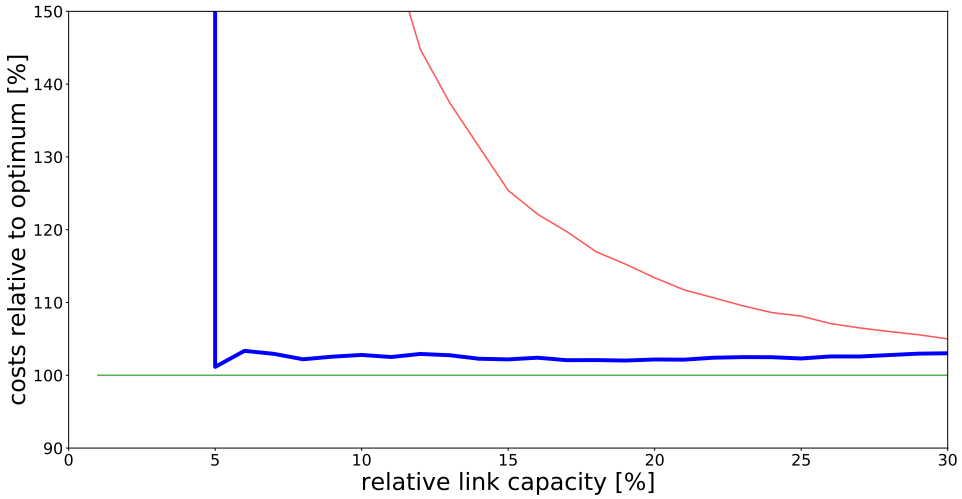

Figure 6.18: Policy 3, trigger function f_1, relative costs

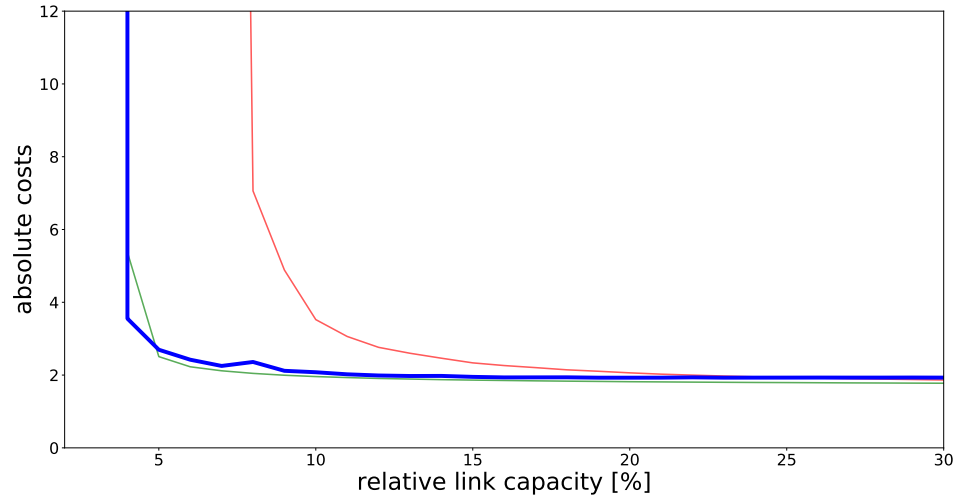

Figure 6.19: Policy 3, trigger function f_2, absolute costs

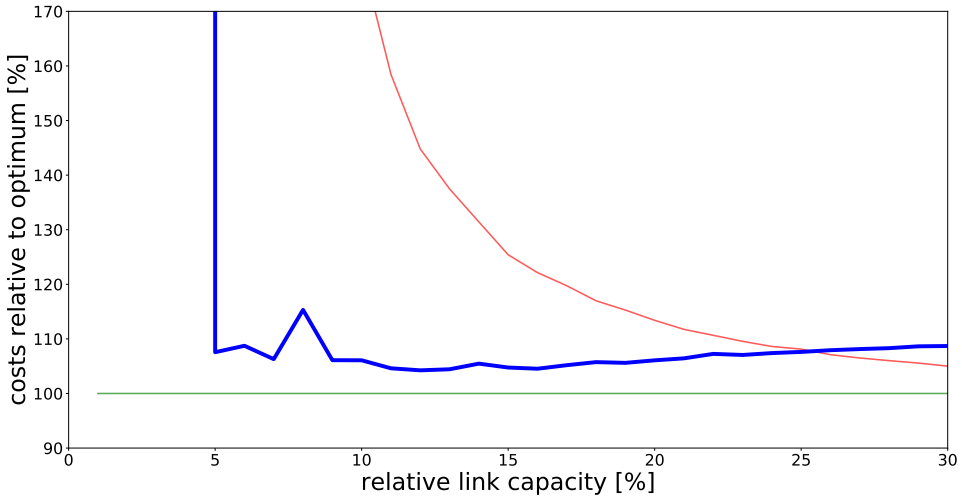

Figure 6.20: Policy 3, trigger function f_2, relative costs

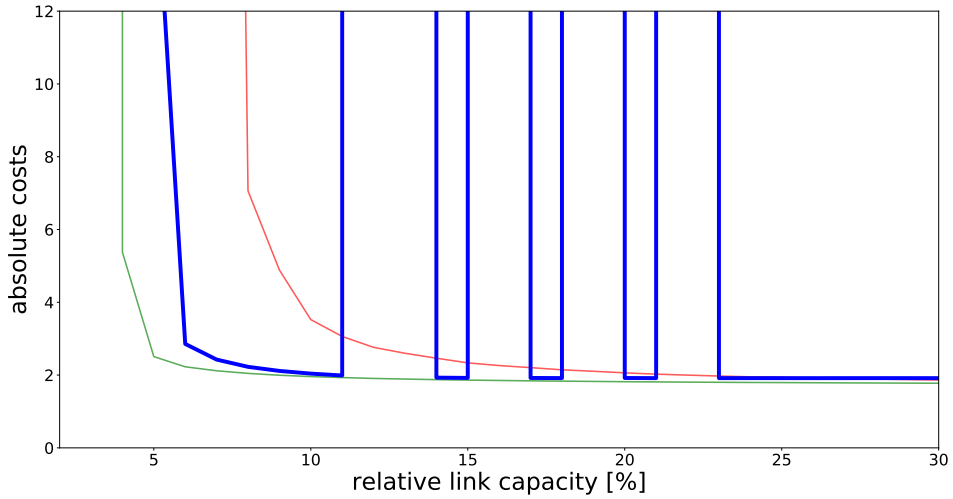

Figure 6.21: Policy 4, trigger function f_1, absolute costs

Of the four policies proposed, this approach is arguably the best: In the critical range with $B_r \leq 30$ %, this strategy offers huge advantages compared to the naive approach. With only about 4 % deviation for $B_r > 5$ %, is just slightly worse than the optimum (using f_1). In the same link capacity interval, the naive approach in turn suffers greatly and performs about 7 % – 200 % worse compared to the optimum. Using trigger function f_2, the general assertion still holds, although with slightly shifted characteristics. The costs of this approach exceed the naive trigger policy in the range above $B_r > 26$ % by about 5 % (w.r.t to the optimum). By careful adjustments of the window size to the scenario at hand, this policy could impact the performance of real-world systems significantly.

6.2.4 Policy 4: Centralized Adjustment in Log-Space

The following figures correspond to the more sophisticated centralized approach:

- Used trigger function: f_1, absolute costs: Figure 6.21.
- Used trigger function: f_1, relative costs: Figure 6.22.
- Used trigger function: f_2, absolute costs: Figure 6.23.
- Used trigger function: f_2, relative costs: Figure 6.24.

The theoretical advantages over its simple counterpart are rendered void due to the same effects, i.e., individual NCS whose costs soar extremely high because of the problem described in Section 4.3.2. Even with a bigger window size to the performance is worse over almost the entire B_r interval investigated. Additionally, with trigger function f_2, the threshold above which this approach performs worse than the naive approach shifts by approximately 14 % to lower link capacities.

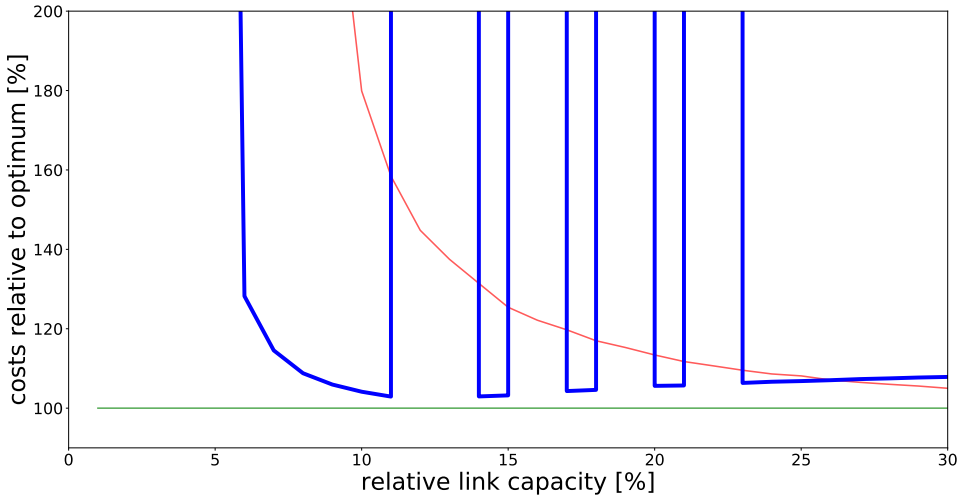

Figure 6.22: Policy 4, trigger function f_1, relative costs

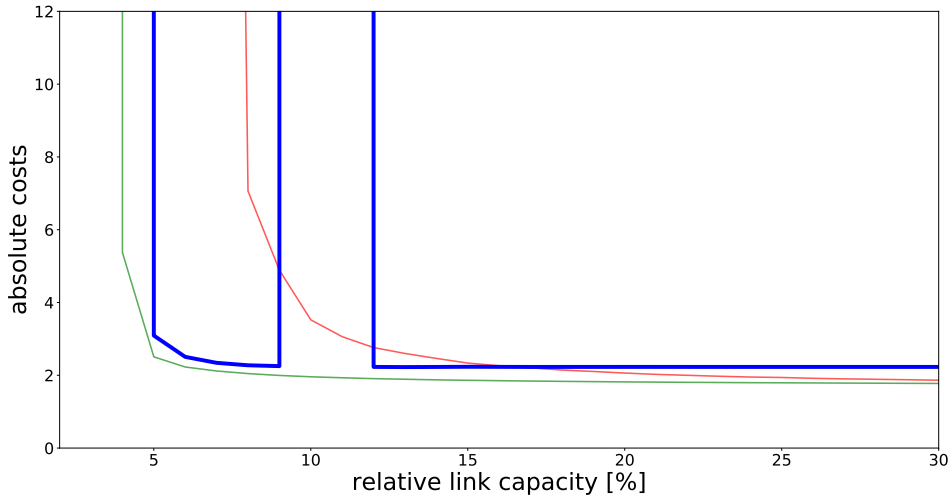

Figure 6.23: Policy 4, trigger function f_2, absolute costs

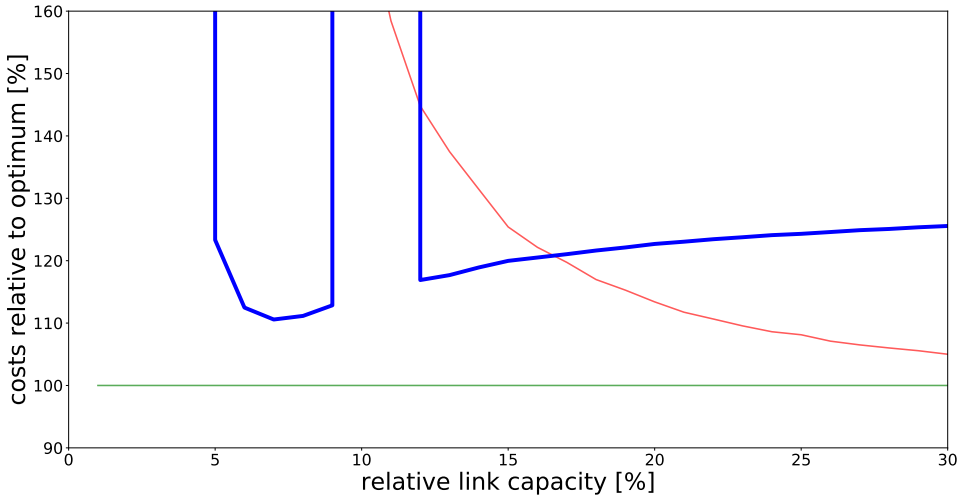

Figure 6.24: Policy 4, trigger function f_2, relative costs

6.2.5 Centralized vs. Decentralized Approaches

As shown in the previous sections, the decentralized approach suffers greatly from single control systems for which the adaption does not work properly. The main reason behind this degradation is the inability of single NCS to reduce or mitigate the influence of noise on the parameter adoption. Even with an increased window size by the factor of 10, this effect remains strong.

Although the centralized approach performs considerably better, it comes with the downside of possible real-world limitations. In real settings, the information used to adapt the trigger parameter c_j must be available to all NCS. This constitutes a serious obstacle, since the availability of this information depends greatly on the ability to transmit and receive packets over the network. However, such a requirement contradicts the goal of minimizing the network load in order to increase the NCS performance by transmitting more control information. Finding the best trade-off between the sharing of control system information (i.e., metrics like performance or trigger rate) and the actual controlling (i.e., input transmitted) over the network may be a subject for future work.

7 Conclusion and Future Work

In this thesis, the optimization of discrete-time event-based control systems communicating over a network is addressed. Previous work on this subject includes various approaches that cover different aspects of a Networked Control System (NCS). These include a method that assigns a static priority to each packet sent. This is implemented by defining flow classes, which are handled at the network infrastructure, in particular at switches. This approach makes use of an Ethernet Extension called IEEE 802.1Qbv, which allows the reservation of resources for specific flows and thus allows to make guarantees about the network connection, including latency boundaries. With these real-time capable networks it is possible to engineer NCS that adhere to specific performance boundaries. Downsides of this strategy include the inability to react (quickly) to external influence and high costs associated with the modification of the reserved resources, since the solution to the No-wait Job-shop Scheduling Problem must be recomputed. An improvement already explored is the setting of the flow priorities not in the network, but at the individual NCS. Although this mitigates the negative aspect of recomputing the flow scheduling and increases the performance, the disadvantage that the flow handling must be implemented and done in the network remains. This is usually accompanied by a significant costs, particular caused by the implementation overhead.

A radically different approach is presented in [LA18]. It uses trigger functions to implement an event-based NCS. While previous approaches address mainly the stability, this work focuses instead on the performance of discrete-time NCS. The contribution of this thesis is the development and evaluation of a method that adapts the trigger condition in a way that optimizes the overall performance of the system.

Different trigger adaption policies were proposed, covering both centralized and decentralized approaches. The general concept is to leverage the quadratic nature of the cost function of the LQR controller used. This was done by using gradient descent as an optimization technique in order to find the cost minimum and thus increase the performance. The policies proposed were assessed by using a simulation framework developed for this purpose. The evaluation showed that the overall costs depend greatly on the link capacity. Furthermore, centralized approaches performed significantly better than their decentralized counterparts. The best results were obtained by the trigger adaption policy 3, which uses a centralized point of view, as well as a heuristic depending on the trigger and drop rate.

The performance reduction in comparison to the optimum (i.e., the system triggers at every step and no packets are dropped) amounts to only about 4 %. However, policy 3 performs a lot better than the naive approach (i.e., the system triggers at every step, packets are dropped). Compared to the optimum, this performance gain amounts to 6 % to over 500 %, depending on the available link capacity.

Future Work

The approach and policies proposed may be the basis of future work. In particular, the following aspects could be investigated further.

- The systems simulated did not suffer from any external disturbances which induce a change of the system state. The control system should normally stabilize the system again. However, it is unknown how the trigger adaption policies behave, if disturbances influence only some but not all of the NCS. In particular, this may affect the performance when using centralized approaches. It is an open question how to design these in order to handle such disturbances.
- The software developed allows the simulation of heterogeneous settings (i.e., those scenarios consisting of multiple but different sets of NCS regarding the trigger function, adaption parameters and control system). An open question is, how the trigger adaption policies perform within such heterogeneous scenarios. It may be possible that the development of a new adaption method is required.
- The window size used to compute averages is currently static. Making this dynamic poses the problem of bad reaction times on, for example, disturbances. Developing a smart window size adaption could solve this problem and improve the performance of a trigger adaption policy further.
- In this thesis, the behavior of two trigger functions was investigated. Several other trigger functions are proposed in [LA18]. It is an open question how their parameters could be adapted dynamically in order to optimize the overall performance.
- As stated in the evaluation (see Section 6.2.5), the applicability of centralized adaption policies may be reduced. This is due to the bandwidth limitations. A trade-off between information about the triggering of other NCS and controlling itself must be found.

Bibliography

[Ast08] K. J. Aström. “Event based control”. In: *Analysis and design of nonlinear control systems*. Springer, 2008, pp. 127–147 (cit. on p. 16).

[CBDR17] B. W. Carabelli, R. Blind, F. Dürr, K. Rothermel. “State-dependent priority scheduling for networked control systems”. In: *2017 American Control Conference (ACC)*. May 2017, pp. 1003–1010. DOI: 10.23919/ACC.2017.7963084 (cit. on p. 16).

[DN16] F. Dürr, N. G. Nayak. “No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: ACM, 2016, pp. 203–212. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997494. URL: http://doi.acm.org/10.1145/2997465.2997494 (cit. on p. 15).

[HJT12] W. P. M. H. Heemels, K. H. Johansson, P. Tabuada. “An introduction to event-triggered and self-triggered control”. In: *2012 IEEE 51st IEEE Conf. on Decision and Control (CDC)*. Dec. 2012, pp. 3270–3285. DOI: 10.1109/CDC.2012.6425820 (cit. on p. 22).

[HPO16] M. Hermann, T. Pentek, B. Otto. “Design Principles for Industrie 4.0 Scenarios”. In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. Jan. 2016, pp. 3928–3937. DOI: 10.1109/HICSS.2016.488 (cit. on p. 11).

[KPG09] F. Khomh, M. D. Penta, Y. Gueheneuc. “An Exploratory Study of the Impact of Code Smells on Software Change-proneness”. In: *2009 16th Working Conference on Reverse Engineering*. Oct. 2009, pp. 75–84. DOI: 10.1109/WCRE.2009.28 (cit. on p. 42).

[LA18] S. Linsenmayer, F. Allgöwer. “Performance oriented triggering mechanisms with guaranteed traffic characterization for linear discrete-time systems”. In: *2017 European Control Conference (ECC)*. June 2018 (cit. on pp. 16, 22, 24, 29, 31, 32, 34, 69, 70).

[Lee08] E. A. Lee. *Cyber Physical Systems: Design Challenges*. Tech. rep. UCB/EECS-2008-8. EECS Department, University of California, Berkeley, Jan. 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html (cit. on p. 11).

[LL13] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013 (cit. on p. 41).

[MCVG10] P. Martí, A. Camacho, M. Velasco, M. E. M. B. Gaid. “Runtime Allocation of Optional Control Jobs to a Set of CAN-Based Networked Control Systems”. In: *IEEE Transactions on Industrial Informatics* 6.4 (Nov. 2010), pp. 503–520. ISSN: 1551-3203. DOI: 10.1109/TII.2010.2072961 (cit. on pp. 16, 24).

[Mey05] S. Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005 (cit. on p. 48).

[PM02] A. Prodic, D. Maksimovic. “Design of a digital PID regulator based on look-up tables for control of high-frequency DC-DC converters”. In: *Computers in Power Electronics, 2002. Proceedings. 2002 IEEE Workshop on*. IEEE. 2002, pp. 18–22 (cit. on p. 22).

[Tan+03] A. S. Tanenbaum et al. “Computer networks, 4-th edition”. In: *ed: Prentice Hall* (2003) (cit. on p. 15).

[VFM03] M. Velasco, J. M. Fuertes, P. Marti. “The self triggered task model for real-time control systems”. In: *24th IEEE Real-Time Systems Symposium (work in progress*. 2003, pp. 67–70 (cit. on p. 24).

[Wic+11] H. Wickham et al. “The split-apply-combine strategy for data analysis”. In: *Journal of Statistical Software* 40.1 (2011), pp. 1–29 (cit. on p. 48).

All links were last followed on October 5, 2018.

A Appendix: Developer's Guide

This document is intended for users and developers of the software used to simulate event-based control systems. Please refer to the architecture described in Chapter 5.2.1 for a detailed discussion of design decisions. This document is intended as a "getting started" guide only.

A.1 Short Overview

A.1.1 Getting the Source Code

The source code management used is Git. The code is hosted on GitHub at the following address: `https://github.com/Replic4tor/master-thesis-event-based-control`

Note that this serves as a read-only repository, which means that no modifications can be pushed. However, it is allowed and encouraged to clone and fork this repository.

A.1.2 Compiling the Source Code

The build system used is CMake[1]. Although no exotic features are used, the recommended minimum version is 3.5. CMake uses a project description file called "CMakeLists.txt". Based on this configuration file and the environment CMake is installed in, the compiler is selected automatically. Note that your compiler must support C++-11 features. The Gnu C++ compiler supports this from version 4.7.

The compilation has two phases. First, the environment-specific makefiles are written. It is recommended to use an out-of-source build, so that compilation artifacts do not clutter the source code tree. To do this, enter from a command line:

```
user@host:master-thesis-event-based-control $ mkdir build && build
```

Next, create the make files with:

```
user@host:master-thesis-event-based-control/build $ cmake ..
```

The source code could then be compiled with:

```
user@host:master-thesis-event-based-control/build $ make
```

A.1.3 Running the Code

If the compilation was successful, two binaries are generated:

- The simulator itself with the relative path `master-thesis-event-based-control/build/src/ebs`
- The aggregator with the path `master-thesis-event-based-control/build/src/aggregator/aggregator`

[1] https://cmake.org/

These binaries could then be run with with
`user@host:master-thesis-event-based-control/build/src/ $ ./ebs`
and

`user@host:master-thesis-event-based-control/build/src/aggregator $ ./aggregator`
respectively

To simulate a scenario, pass the configuration file as first parameter, e.g.,
`user@host:master-thesis-event-based-control/build/src/ $ ./ebs ../../scenarios/example-scenario.json`

The folder `master-thesis-event-based-control/scenarios` contains some well-documented example scenarios.

A.2 Extension

The extension of the software is straightforward. The following two examples show how to add a new trigger function. Adding a new adaption policy is done in the same way.

A.2.1 Add a Trigger Function

The source code contains a dummy trigger function which serves as a template for new trigger functions.
Essentially, the tasks are

1. Add header and implementation file in `master-thesis-event-based-control/src/triggering`
2. Let the new trigger function class inherit from `TriggerFunction`.
3. Implement three methods:
   - `get_id()`. This returns a number that is used to identify the trigger function in the results (CSV files).
   - `get_initial_state()`. This returns the initial state of the trigger function.
   - `trigger_decision_nvi()`. This is the core of a trigger function. Note to handle any state thread-local in order to avoid interferences from other threads.
4. Add the new trigger function in the file `master-thesis-event-based-control/src/EbsConfig.cpp` by extending the `parse_trigger_function()` method.

After these steps, the newly created trigger function could be used by specifying its id in a scenario file.

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