Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 83

In-network Packet Priority Adaptation for Networked Control Systems

Stephan Zinkler

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M. Sc. Naresh Ganesh Nayak, Dipl.-Ing. Ben Carabelli
Commenced:	1. March 2016
Completed:	31. August 2016
CR-Classification:	C.2.3, C.2.6

Abstract

Sharing the network between Networked Control System (NCS) having strict demands with respect to latency and jitter and applications only requiring best-effort service leads to multiple problems. An important task to consider is how to prioritize individual types of traffic in such a way that the necessary guarantees for an NCS to be stable can still be given. While there are ways to prioritize the more important control traffic of an NCS over best-effort traffic sharing the same network, a more sophisticated approach has to be found in order to handle multiple NCS sharing the highest priority. In this thesis, in-network priority scheduling applications with a global view on the network are developed in order to schedule and prioritize individual NCS such that their stability can be guaranteed while sharing the network between multiple NCS.

This thesis deals with in-network packet priority scheduling for Networked Control Systems. Using Data Plane Development Kit (DPDK) to achieve a Network Function Virtualization (NFV) based approach, a priority scheduling application is implemented in a middlebox to handle continuous priorities. This application could be instantiated and migrated within the network while simultaneously using Software Defined Networking (SDN) to route the traffic to the respective nodes. Additionally, this approach is extended using SDN and OpenFlow to adapt priorities in-network. Using the eight internal per-port queues of a switch, discrete priorities are used to schedule, and additionally adapt, the priorities on the switch. This approach could give the opportunity for priority-based routing by using the SDN-controller for routing decisions and configuring the switches.

The evaluation of this thesis is done by simulating NCSs and emulating the network containing the middlebox. For this, a simulation of an inverted pendulum is implemented for which the use of DPDK is compared to standard sockets. It can be shown that DPDK is able to perform better due to less delay and jitter. The scheduling application is evaluated by comparing it to a round-robin scheduling approach. The result suggests that the application is able to keep multiple NCS more stable than it's round-robin counterpart. Furthermore, it is able to stabilize a more unstable system faster and more effectively. While the maximum sampling time for a system with a pendulum having an initial angle of 35° was found to be 50ms for the round-robin scheme, the middlebox is able to keep the system stable until 120ms. The application using OpenFlow is evaluated with respect to the time it takes to configure the switch as well as the overhead imposed by the configuration compared to the number of NCS within the network.

Acknowledgement

I would first like to thank everybody who supported, motivated and helped me in the course of this master thesis. This accomplishment would not have been possible without all of them.

This especially applies to both my supervisors, Ben Carabelli and Naresh Nayak, of the Insititue for Parallel and Distributed Systems (IPVS) at the University of Stuttgart, who guided me through the past months. The door to their offices was always open whenever I ran into difficulties or had any questions about my research or writing. Without their dedication, guidance, and frequent reviews, this report would not be as it is now. Additionally, I would like to thank Dr. Frank Dürr, who helped me overcome the one or the other problem, as well as Prof. Kurt Rothermel for providing me with the opportunity to contribute to this research.

Further thanks goes to the people who spend their time proofreading this work, pointing out mistakes and showing me where there was need for clarification. In addition, I want to thank my friends and girlfriend for providing me with the much needed support. Finally, I must express my profound gratitude to my parents for their unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

DPDK Data Plane Development Kit

DPI Deep-Packet-Inspection

DSCP Differentiated Services Codepoint

EAL Environment Abstraction Layer

IoT Internet of Things

NAT Network Address Translation

NCS Networked Control System

NFV Network Function Virtualization

NIC Network Interface Card

PCP Priority Code Point

SDN Software Defined Networking

TOS Type of Service

VLAN Virtual Local Area Network

1 Introduction

In many modern industrial and commercial manufacturing systems, computing, communication and control is often integrated into different factory operations and processes [1]. NCS are at the heart of automation technologies used in these systems. In NCS, control loops are closed over a communication channel, allowing for cost-effective and flexible applications [2]. Due to steadily decreasing price, increasing speed, more and more software and applications being available, and a well-established infrastructure, the field of Networked Control Systems has been growing steadily for the past decade. Such systems can nowadays be found in various areas, including aerospace, automotive, transportation, manufacturing, or civil infrastructure, thus clearly showing its value and importance as a key technology [3].

An NCS is a distributed system where a physical system (the plant) is controlled using sensors and actuators (directly connected to the plant), which communicate with the controller over a real-time network. These Networked Control Systems heavily rely on the communication channel since the control traffic sent over this network is highly time-sensitive. Therefore, strict guarantees with respect to latency and jitter are needed from the underlying communication network to provide stability for the system. In many cases, field-bus networks such as CAN, which gives timing-guarantees for high priority messages, are used to achieve sufficient guarantees. However, with the growing significance of the Internet of Things (IoT), applications may share the network with control systems, thus sending time-sensitive traffic of the NCS along with non-time-sensitive traffic of other applications. While network technologies such as field-buses are practical for specific scenarios, their usability in combination with applications that usually run on commodity networks such as IP or IEEE 802 protocols, as used on the Internet, is limited and may result in higher costs and inferior efficiency.

However, these networks only provide best-effort services and give no guarantees regarding timeliness whatsoever, therefore posing a problem for Networked Control Systems. To cope with the loss of these guarantees, other mechanisms have to be employed. One approach is to use already existing mechanisms in the IP stack to prioritize the NCS traffic over none-time-sensitive traffic. As a naïve approach, the sender of time-sensitive traffic, for instance a sensor sending its data to the controller, sets a specific field in the packet to mark it as time-sensitive high priority traffic. Upon receiving the packet, the network would automatically prioritize marked high priority messages over unmarked non-time-sensitive ones, allowing the NCS traffic to traverse the network without being impeded by regular traffic. Most commodity switches can be configured to prioritize packets according to the packet's header fields such as the Priority Code Point (PCP) of Virtual Local Area Network (VLAN) tags or Differentiated Services Codepoint (DSCP), thus supporting this kind of prioritization.

While this naïve approach is a good way to decouple time-sensitive from best-effort traffic, it does not solve the aforementioned problem entirely. Although the interference of non-time-sensitive traffic can be mitigated, the network does not handle multiple packets from multiple Networked Control Systems arriving at the network at the same time. This simultaneous burst arrival of high priority traffic could cause increased latency and jitter since each NCS packet would be considered equally important and queued for further transmission, thus causing a congestion. Additionally, this obstruction could result in delays or packet drop, thereby violating the necessary guarantees to provide a stable system. Therefore, a network being used by multiple NCS has to find a way to handle traffic of multiple NCS.

One approach could be to schedule the time-sensitive traffic accordingly in order to adhere the necessary timing guarantees of each individual Networked Control System. However, distributed mechanisms in conventional networks still do not solve congestion at specific nodes. What is needed is a centralized approach to manage a global view on the network, thus being able to prioritize individual NCS and their traffic to provide guarantees with respect to arrival time.

In recent years, SDN has emerged as a technology to separate the control (e.g. routing decisions) from data plane (i.e. forwarding). This is done by introducing a logically centralized SDN-controller, which handles policy enforcement and network-wide traffic forwarding decisions, while routers and switches within the network are only used as forwarding devices [4, 5]. With the SDN-controller having a global view on the network, SDN could now be used as a way to enforce scheduling of time-sensitive traffic of an NCS within the network. However, standard commodity switches do not support scheduling according to priorities. What is needed are designated middleboxes responsible for this kind of scheduling that could be distributed throughout the network.

To achieve flexibility and scalability, these middleboxes could be designed using NFV, allowing for versatility with respect to instantiation and relocation. These virtual network functions could then be migrated to suitable places within the network without imposing overhead with respect to dedicated hardware requirements. An SDN controller would then configure the network such that the respective traffic is routed through the location of the virtual middleboxes, giving it the desired global view of the entire network.

The objective of this thesis is to implement the above mentioned methods to create a centralized view on the network, thus being able to handle multiple NCS using the same network. More precisely, the goals are as follows:

- Providing a centralized mechanism for the scheduling of multiple NCS within the same network
- Developing approaches to schedule time-sensitive traffic along with best-effort traffic
- Implementing these approaches in a scalable manner with minimal overhead
- Giving a proof of concept using SDN as a scheduling technique within the network
- Implementing a simulation of a NCS for evaluations

1.1 Structure

In the further course of this thesis, the individual chapters are structured as follows:

- **Chapter 2 - Background and Related Work:** In this chapter, the basics of Software Defined Networking, Networked Control Systems and Network Function Virtualization are described. In each section, the correlation to this thesis is shown.
- **Chapter 3 - System Model and Problem Statement:** The system model, the architecture and the individual parts of the system are subject to this chapter.
- **Chapter 4 - Contribution:** The focus in this chapter lies on the implemented scheduling applications to fulfill the previously stated goals.
- **Chapter 5 - Networked Control System Simulation:** This chapter shows the implementation details of a simulation of a Networked Control System used for evaluation purposes.
- **Chapter 6 - Evaluation:** The previously mentioned simulation is used to evaluate the applications described in chapter 4 to compare the results to the desired behavior.
- **Chapter 7 - Conclusion:** In the end, the objective of the thesis and the achieved results are briefly summarized and a short prospect of future work is given.

2 Background and Related Work

In this chapter, the background and related work for this thesis are elaborated. First, NCS and their mechanics are explained. Second, the reason behind SDN is discussed, the functionality of an SDN network is shown, and OpenFlow, a protocol for SDN, is pointed out. Finally, NFV and packet processing frameworks such as DPDK are examined.

2.1 Networked Control Systems

In many fields such as manufacturing plants, automotive, air- and spacecraft industry, civil infrastructure or transportation, information and control signals are often exchanged between multiple participants such as sensors/actuators, controller, supervisor tasks, or other I/O devices over a communication channel.

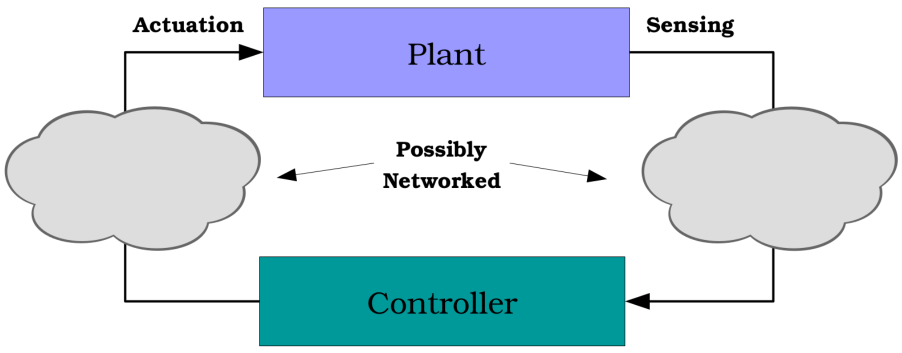


Figure 2.1: Illustration of a Control System. The Controller is connected to plant containing of the actuators/sensors.

These systems are called Networked Control System when the feedback control loop is closed over a real-time network used for the transmission of highly time-sensitive control messages [3, 6, 7]. In recent years, these NCS increasingly gained popularity due to reduced installation and maintenance cost, reduced weight and power requirements, a well established infrastructure, higher reliability and flexible architectures [3, 7–9].

However, using a network imposes additional, often unknown and variable, delay, jitter and packet-losses. Since Networked Control Systems use a timing model where packets have to arrive within the predefined sampling time, these attributes are crucial for the modeling of an NCS. Furthermore, when designing a model for the control system, the network has to be modeled as well in order to cover all aspects of the system. Depending on the delay/loss model used, more or less strict real-time requirements are needed. These requirements can be represented by two different types of NCS models: deterministic and probabilistic.

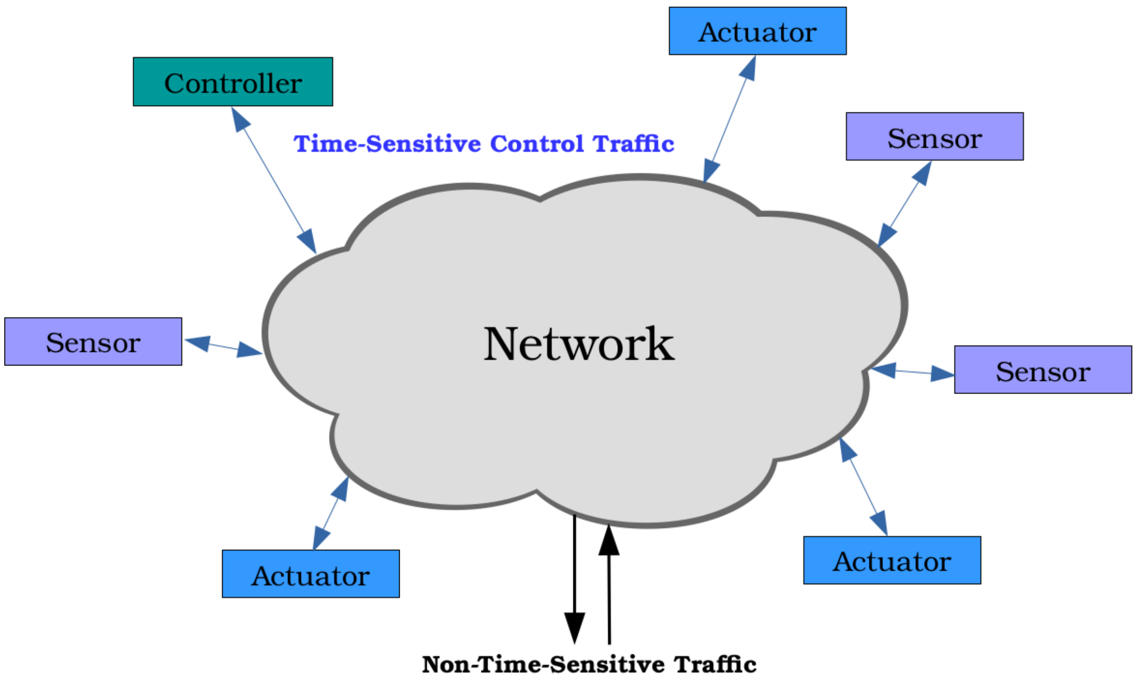


Figure 2.2: Illustration of an NCS. The Controller is connected to the actuators/sensors over a communication network. All control messages are sent and received over this network. The network may be used for additional non-control (non-time-sensitive) traffic.

Deterministic models require known and fixed upper-bounds on delay and do not tolerate significant amounts of loss, thus limiting the used network to one giving rigorous guarantees regarding the real-time capabilities on the data link layer. Examples for these networks are field-buses such as CAN or Ethernet-based networks like EtherCAT [10]. However, with the growing importance of the IoT, applications may share the network with control systems. While network technologies such as field-buses are practical for specific scenarios, their usability in combination with applications that usually run on commodity networks such as IP or IEEE 802 protocols, as used on the Internet, is limited due to higher cost and lesser efficiency. These commodity networks only provide best-effort services with respect to latency and losses, thus posing new challenges for the control systems [7, 11]:

- **Non-control related traffic:** Time-sensitive control traffic is sent over the same network as non-time-sensitive best-effort traffic.
- **Delay:** Sending over a network and exchanging data imposes an often variable and unknown delay and/or jitter.
- **Unreliable transmission:** Packets sent over a network may be lost.
- **Network Constraints:** Bandwidth and packet size limit the network.

All this may limit the performance of or destabilize an NCS without careful consideration and management of the network. For that reason, Networked Control Systems should take network characteristics into account. Delay, jitter and missing packets are very important factors for the stability of an NCS. When sending time-sensitive along with non-time-sensitive traffic over the same network without prioritizing the respective packets, the more important control traffic might get delayed or dropped. In addition, the size of the sent packets and the bandwidth provided by the network are important factors influencing delay and drop rate [7, 11].

Furthermore, due to the lack of real-time guarantees, a deterministic model is, in general, no longer suitable for usage in such networks. Probabilistic models, however, can in principle handle best-effort services provided by the underlying network as long as the delay and loss can be described with an appropriate stochastic model such as a Bernoulli process for the loss of packets [9, 10]. For that reason, this work uses a probabilistic model where a constant network delay and arrival probability of packets is assumed. For more details, see chapter 5 on page 27.

2.2 Software Defined Networking

The management of modern networks is very challenging. Computer networks are typically built from a large amount of switches, routers, and diverse middleboxes. Furthermore, many different and complex protocols are used within the network for transportation or traffic engineering. These constantly evolving and dynamic developments pose a challenge for network operators since they have to manage and configure their networks according to a wide range of network events and policies. In addition, the network has to respond to faults, additional load, and changes, which leads to a cumbersome reconfiguration of the network. This reconfiguration is not only necessary because control (e.g. routing decisions) and data plane (e.g. packet forwarding) are bundled together, but also due to the fact that each network device has to be configured separately using vendor-specific low-level commands [4, 5, 12].

As a result of the problems and limitations of previous approaches to network management, SDN has emerged in a bid to facilitate the management of the network infrastructure. In SDN, an SDN-controller (In the following only referred to as controller) dictates the overall network behavior, while switches and routers only operate as simple forwarding devices. This approach separates the control (controller as the control logic) and data plane (switches and routers). The control logic is now implemented in a logically centralized controller, giving it a global knowledge of the network state, thus simplifying policy enforcement and reconfiguration by making network-wide traffic forwarding decisions [4, 5]. This behavior is especially useful for applications requiring a global or centralized view on the network instead of a distributed one. As previously mentioned, a network being used by multiple NCS requires centralized control, making SDN a perfect candidate for implementing scheduling procedures.

Figure 2.3 shows a simple example of an SDN architecture. The data plane, which consists of the forwarding elements, is connected to the control plane by a southbound interface. The interface is designed according to the following requirements [13]:

- Support flexibility to easily change control schemes
- Allow abstraction of the characteristics of physical resources
- Allow for virtual networks and provide secure isolation between them
- Abstract information regarding the physical network resources

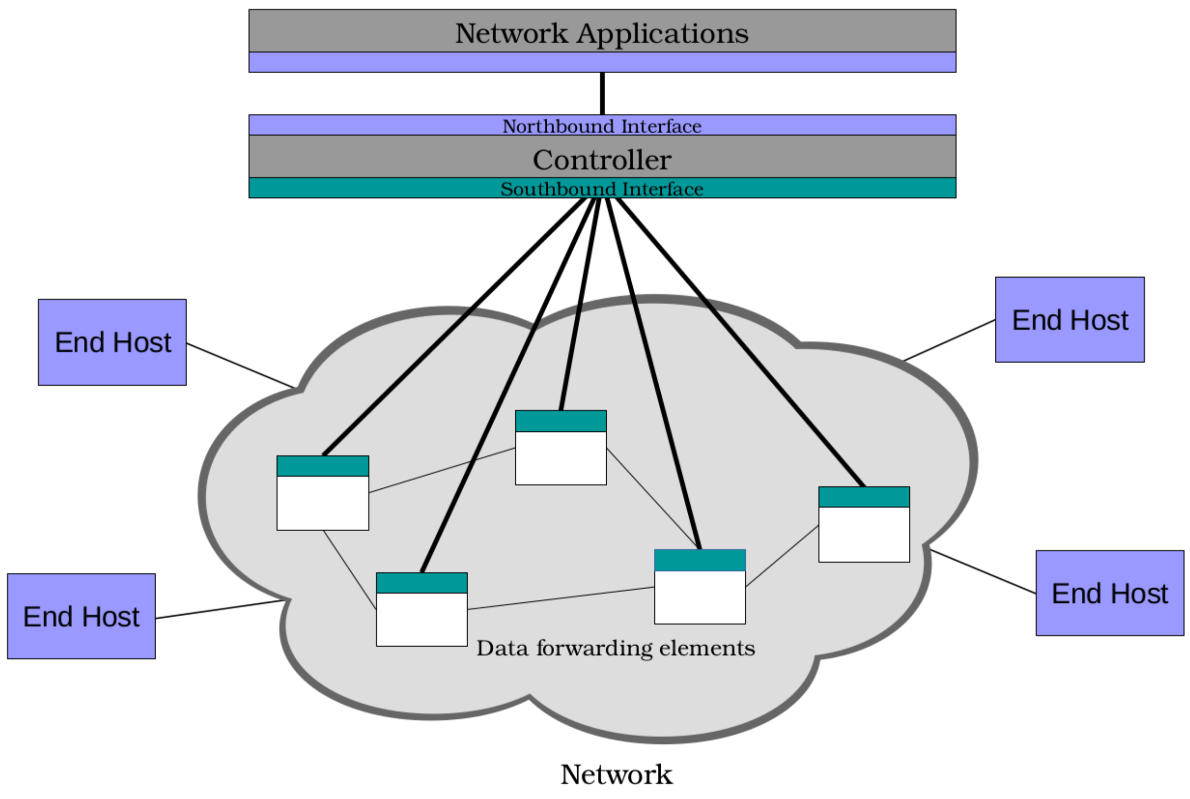


Figure 2.3: Illustration of a simple SDN architecture. The controller is connected with the control logic (network applications) over the northbound interface and to the data plane (network, data forwarding elements) via the southbound interface. Note that the controller is logically centralized, there could be more than one controller instance (Synchronization needed).

The logically centralized controller, on the other hand, is connected to network applications and services implementing the control logic by a northbound interface. This interface is in turn responsible for exposing network related information such as topology, network statistics, and interface functions to the control logic and is defined by the controller. While there exists no standard northbound interface, which is thus defined individually for each controller, the southbound interface is commonly implemented using the OpenFlow protocol.

2.2.1 OpenFlow

A separation of control and data plane, as proposed by Software Defined Networking, calls for the possibility to configure switches on the fly using open/standardized interfaces instead of expensive and complex vendor-specific commands or hardware [4, 5, 12]. OpenFlow has the objective to offer high performance along with low-cost implementations and allow users to run experiments on heterogeneous routers and switches without the need of exposing the internal workings by the vendor or programming complex vendor-specific software [14]. This is done by exploiting a common feature of many modern Ethernet switches and routers: flow-tables. Although most vendor's flow tables are different, McKeown et al. [14] identified several functions common to many switches and routers. This set of functions is exploited to achieve the desired objectives of OpenFlow and support partitioning of traffic into flows.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2.1: Main components of a flow table entry

A flow-table in an OpenFlow switch consists of one or multiple flow entries. Each entry in turn contains the following (see Table 2.1) [15]:

- **Match fields:** Used for matching against packets. Whenever a packet arrives at the flow table, the defined match field entries are compared to that of the packet. Examples for match fields header fields present in Layers 2–4 such as Ethernet source/destination address, VLAN PCP, IPv4 or IPv6 source/destination address or DSCP/Type of Service (TOS) bits.
- **Priority:** Describes the matching precedence of the flow entry. Used if the packets matches multiple table entries.
- **Instructions:** Used for modifying how the packet needs to be processed. Examples of instructions are Goto, which forwards the packet to another table or output, that sends the packet out of the specified port.

An OpenFlow switch can have multiple flow tables, each having their own set of table entries and instructions, which are pipelined. Whenever a packet is received, it is matched against the match field of the first flow table. If the packet is successfully matched, the instruction set corresponding to the respective flow table entry is executed (for instance go to table or add action to action set). In most cases, the instruction will direct the packet to the next flow table responsible for the matched packet, where in turn the packet is matched again. Figure 2.4 illustrates the packet flow through the processing pipeline of an OpenFlow switch having multiple flow tables [15].

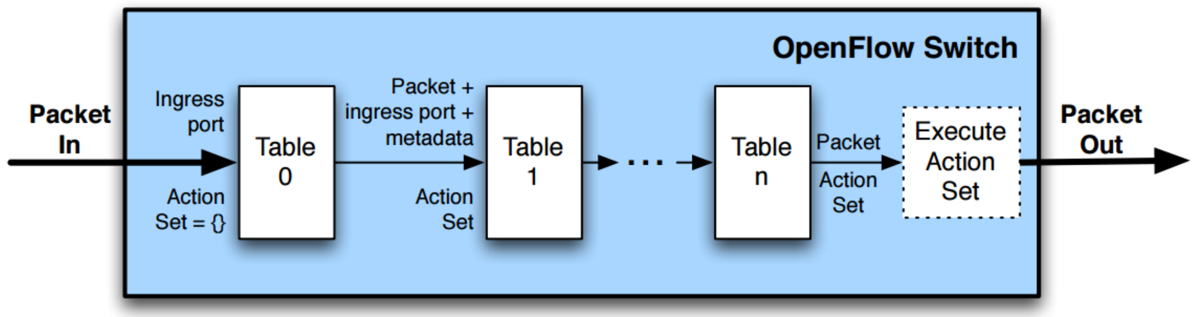


Figure 2.4: Illustration of a OpenFlow Switch having multiple flow tables [15]. When a packet is received, it is matched against the flow table entries of the first flow table and the instruction set included in the matched entry is executed.

When a flow entry does not direct the packet to another table, the pipeline processing stops, in which case the action set corresponding to the packet is executed. If however the packet does not match any entry of the flow table, the packet could, depending on the configuration of the flow table and the table-miss entry, be dropped, passed to another table, or sent to the controller [15]. The exact behavior when matching a packet to the individual match-fields of the flow tables is shown in Figure 2.5.

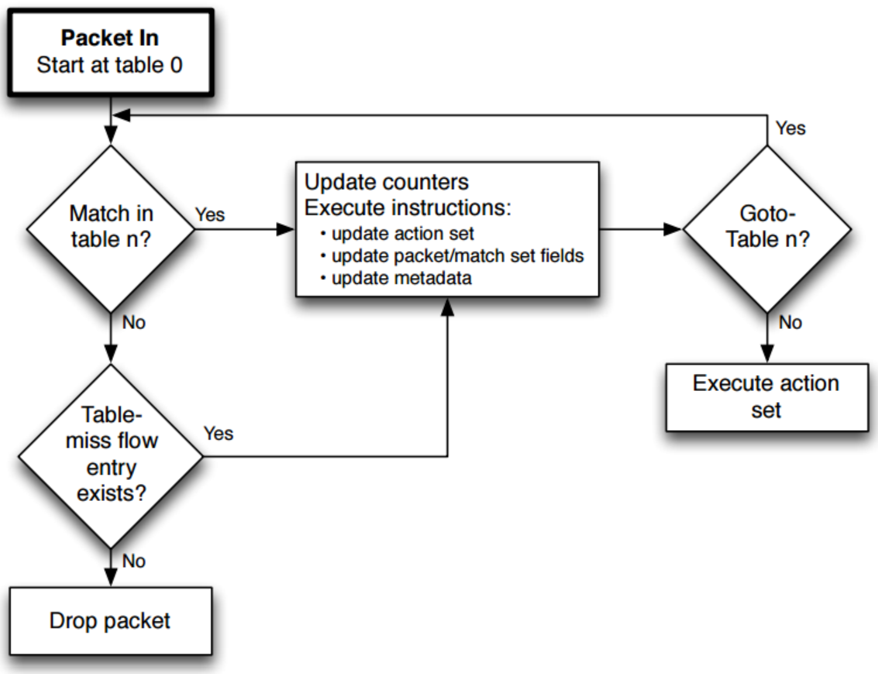


Figure 2.5: Flowchart detailing the packet flow in an OpenFlow Switch [15].

In chapter 4, a controller module is designed using the OpenFlow protocol for generating multiple flow tables and entries by exploiting the pipelining feature at designated switches in order to route packets according to certain criteria. For more details, see page 21.

2.3 Network Function Virtualization

Another recently emerging technology, which is often mentioned together with SDN, is NFV. Deploying new services in today's networks has evolved to be increasingly difficult due to the diversity of devices, equipments, and functions. Not only are skilled professionals needed to integrate and maintain these services, but also the space and energy for these diverse middleboxes [16, 17]. NFV is using the virtualization technology by offering network functions, which are decoupled from the physical network devices. These virtualized network functions can now be executed on commodity hardware, instantiated on demand, and relocated to different parts of the network without changing the hardware or the network itself [16, 17].

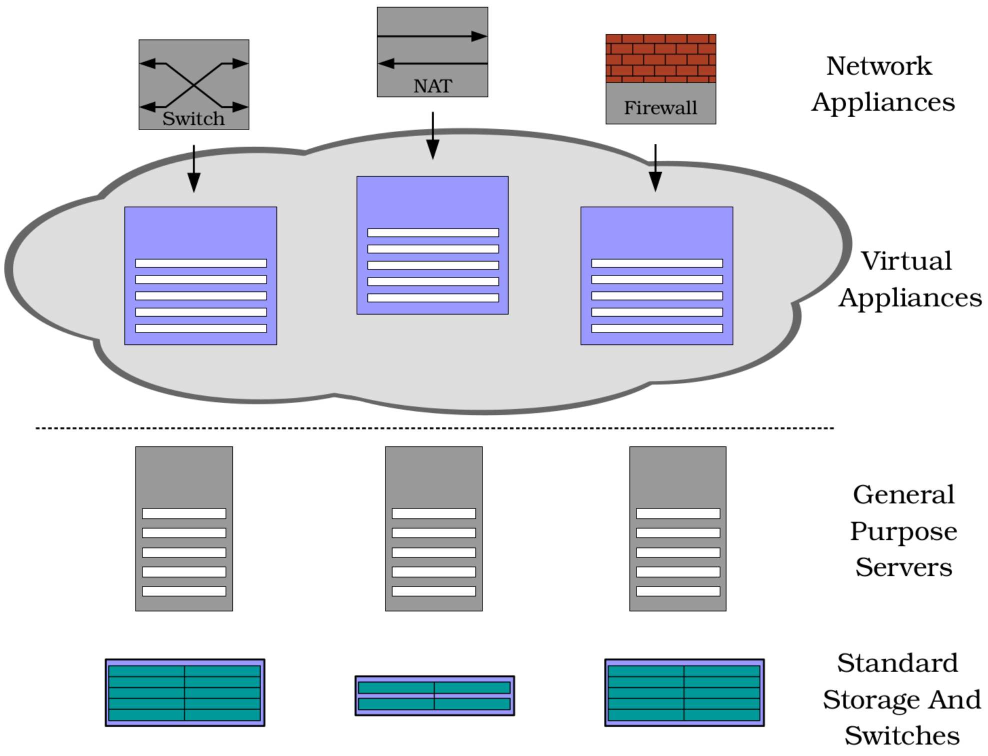


Figure 2.6: Illustration of typical network appliances implemented as NFV's running on commodity hardware such as standard servers, storage and switches [17].

With this technology, a given service could be implemented using a set of virtual network functions, which are in turn run on one or multiple commodity servers. This service could then be relocated on the fly in case of network changes or in order to apply location-based targeting of customers [16]. Further examples implemented as virtualized network functions, as can be seen in Figure 2.6, include switches, Network Address Translation (NAT) or firewalls, which can then be executed on general purpose servers.

In traditional networks, changes in location, load, or it's functions would often result in additional costs since services such as a firewall are regularly integrated using specialized hardware. If, for instance, the network would be used by additional entities, supplementary hardware would have to be bought and installed to handle the increase of load. By using a firewall implemented as a virtualized network function, however, the service could be installed on any node within the network without explicitly buying and adding additional hardware. Furthermore, the complete network could relocate on different servers without having to migrate the physical hardware of the services within the network, thus providing a cost-effective, flexible, and scalable alternative.

2.3.1 Packet Processing Frameworks

Using commodity hardware in combination with virtualized network functions instead of specialized hardware, however, may lead to increased latency variations as well as to significant throughput and performance drops [17, 18]. In addition, it is common practice to use sockets for most types of communication since they abstract the network using a simple interface. However, sockets also impose additional delay and/or jitter on account of the TCP/IP stack. On the other hand, delay, jitter, and the overall performance of nodes within the network are an important aspect for the stability of a NCS, such that different approaches for the transmission and processing of traffic on these nodes have to be found.

One approach to decrease delay and jitter, thus being able to effectively use commodity hardware, is the DPDK Framework. DPDK has been developed by Intel and is a set of libraries and drivers designed for fast packet processing. The framework creates these libraries for specific hard- and software environment by using an Environment Abstraction Layer (EAL) that provides the interface to the library as well as hardware accelerators. Furthermore, DPDK includes optimized Network Interface Card (NIC) drivers and ring buffers, used for sending and receiving packets, to achieve faster packet processing [20].

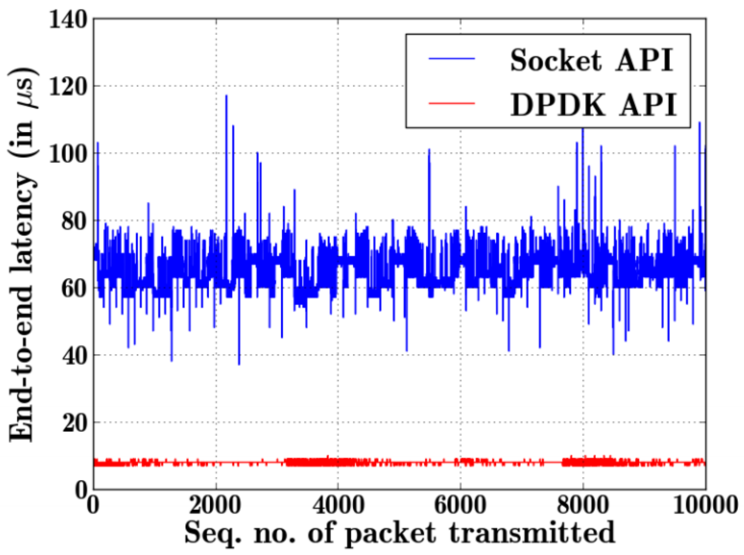


Figure 2.7: Latency of Sockets vs. DPDK [19]

Instead of using interrupts, DPDK employs polling on the ring buffers, thus eliminating the overhead of context switches imposed by the interrupt handling. However, this procedure implies a 100% utilization of the respective task running the polling mechanism. To cope with this characteristic, DPDK pins each individual process to a dedicated processor core. However, this also means that each DPDK task will completely occupy one core such that it cannot be used for other tasks. In addition, the number of possible tasks is limited by the number of processor cores on the system running DPDK.

Using DPDK, the delay and, more importantly, the jitter can be decreased significantly compared to an application running on sockets (see Nayak, Dürr, and Rothermel [19]). This property is utilized in chapter 4 where a DPDK application in combination with a NCS is implemented (For further details see page 19).

Alternatively, instead of DPDK, the approach of Rizzo [21] could be used to achieve the same goal. In this work, features such as preallocated, fixed size packet buffers, multiple hardware queues, and direct access to protected packet buffers are used to achieve high performance and fast packet processing.

3 System Model and Problem Statement

This chapter first describes the system model of a communication network used by Networked Control Systems. Afterwards, the problem statement tackled in this thesis is elaborated.

3.1 System Model

NCS require strict timing guarantees and deadlines on the latency of control traffic sent over a bandwidth-limited network. One way to achieve in-time delivery of critical Networked Control Systems is to prioritize the control traffic of the respective NCS over the remaining time-sensitive traffic.

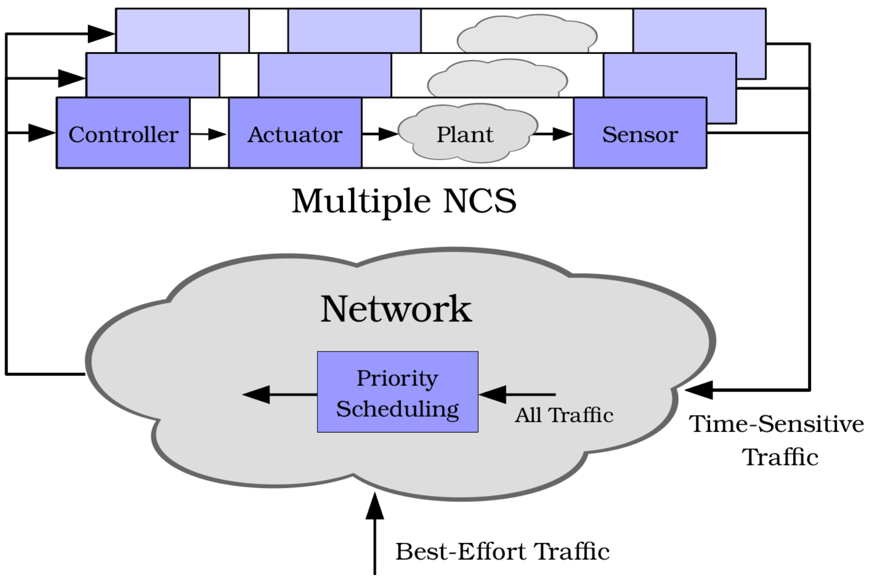


Figure 3.1: Illustration of the system architecture. The network is used by multiple distributed control systems. The sensor is sending system variables over the network to the controller, which in turn is calculating updates. The updates are sent to the actuator, which is in close proximity to the controller.

For this, priorities, or more specifically costs, have been calculated by a designated entity and added to each control traffic packet. These priorities can then be used to schedule the more important packets having higher priorities before the remaining ones. In the model used in this thesis, both continuous and discretized priorities are used by different applications.

The controller and sensor parts of the NCS are connected over a real-time IP network. The sensor's status update is sent to the controller over this network. For the sake of simplicity, we assume that the actuator is in close proximity of the controller such that the values calculated by the controller are not sent back over the network but updated immediately within the system. All traffic of each individual NCS is sent over the same network. Within the network, there are switches used for scheduling the control traffic of each NCS according to preassigned priorities. The algorithms and methods used for these scheduling procedures are part of this thesis.

3.2 Problem Statement

This thesis focuses on the communication channel of an NCS. In previous work, VLAN-tags or the DSCP field were used to prioritize the traffic within the network [22–24]. While this approach could also be used to prioritize the control-traffic over regular non-time-sensitive best-effort traffic, it does not necessarily guarantee that packets belonging to the control traffic arrive in time at the controller/actuator. Although best-effort traffic no longer interferes significantly with the time-sensitive traffic, multiple NCS packets arriving at the same time at a node within the network do. To cope with that problem, this thesis proposes a priority scheduling mechanism within the network. This mechanism's task will be to separate best-effort from control traffic while at the same time using the packets priority, preassigned by the control system's sender, to enforce an order of outgoing control packets.

Assume that the bandwidth of the network is limited to b $[\frac{bit}{sec}]$, the sampling time of the control system is represented by T_s $[sec]$ and the size of each control packet is S $[bit]$, where each packet consists of the measurement (y_i) and the priority P_i. Then the number of possible control packets C that could be sent within one sampling time T_s can be calculated by

$$C = \left\lfloor \frac{T_s * b}{S} \right\rfloor \tag{3.1}$$

The applications task is now to schedule the incoming packets such that traffic of the C highest priority NCS is delivered in time. Figure 3.2 illustrates the procedure.

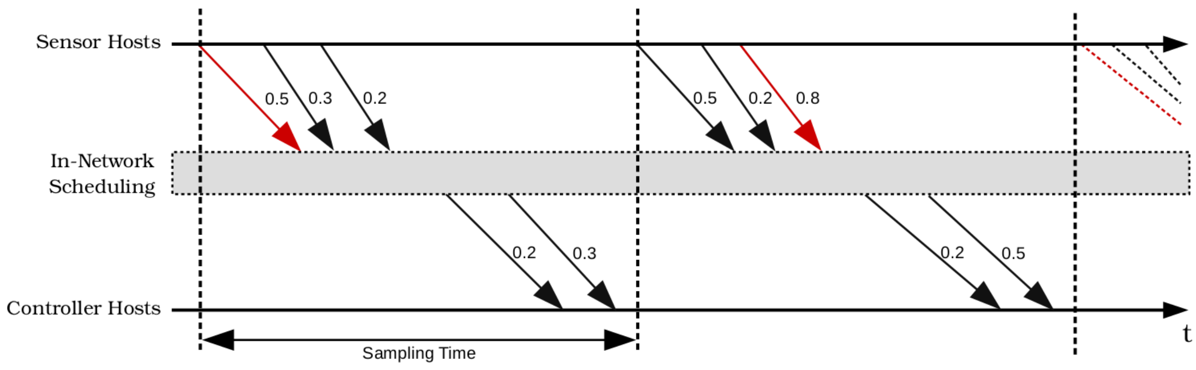


Figure 3.2: Illustration of multiple NCS sending traffic over the network. Since only two of the three packets sent can be delivered before the next sampling time, an application has to manage the scheduling of critical NCS (The lower the priority value, the more important the packet). In this example, the packets having priorities of 0.2 and 0.3 respectively are sent in the first, 0.2 and 0.5 in the second sampling time, whilst 0.5 and 0.8 are dropped.

There are three control packets sent from the sensors. As can be seen, only two of these packets can be delivered before the next sampling time. To guarantee that the C highest priority NCS are stable, these packets have to be sent first, thus justifying the presence of a scheduling application within the network. As the example shows, an in-network scheduling application would forward the packets with priorities 0.2 and 0.3 in the first sampling period while dropping the packet with a cost of 0.5.

With this approach, assumptions regarding timeliness of the traffic belonging to the NCS having the highest priority can be made since these packets will always be sent out first. Furthermore, the higher the priority, the more likely an in-time delivery. The goal of this strategy is to maximize the number of stable NCS within the network.

4 Contribution

This chapter focuses on the work done in order to achieve the goals described in chapter 3. At first, a priority scheduling application using DPDK is introduced and explained in detail. Furthermore, an extension to priority scheduling is elaborated using SDN and OpenFlow to change and/or adapt priorities while passing through a switch.

4.1 Priority Scheduling

To create a global view of the network, thus being able to prioritize control traffic according to delay, jitter, and loss, a centralized application is needed. Since the priority of each individual packet is calculated by the NCS's sensor and is thus continuous (for details see chapter 5), a designated middlebox is needed to schedule the traffic according to these continuous priorities. This middlebox has been implemented using DPDK and is executed on a separate host within the network (see https://github.com/ZnSn/Masterthesis.git). The application is constructed as follows:

- There are three processes, each running on a designated core.
- Inter process communication is done via shared memory.
- The first process is used for setting up the application and DPDK.
- One of the remaining two processes is used for sending, the other for receiving.
- The application is using two ports connected to the DPDK drivers. The receiver process is linked to the ingress port, the sender process to the egress port.

Whenever a packet arrives at the host's interface, the packet's header is checked to distinguish best-effort from time-sensitive traffic. This is done by examining a predefined header field such as TOS, VLAN, or the destination address. Depending on the classification of the packet, one of two paths are traversed. In case of a best-effort traffic packet, the packet is put straight into a FIFO queue for later dispatch. If however the packet belongs to a time-sensitive NCS traffic, a Deep-Packet-Inspection (DPI) is carried out to differentiate between individual NCS or their priorities respectively. This is done by reading the packets payload, which holds the (continuous) priority of the packet. According to the extracted priority, the packet is put into a priority queue at the respective position (Where a low value means a high priority). This is all done at the receiving part of the application (see Figure 4.1).

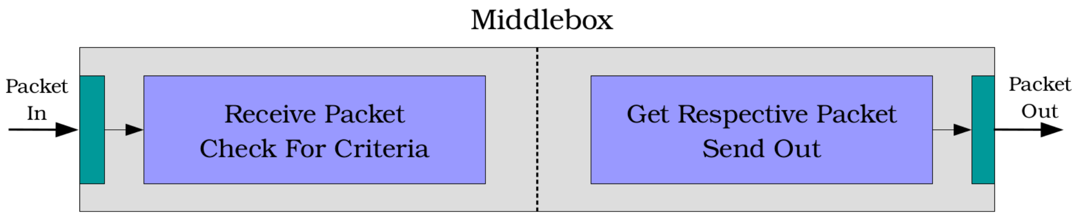

Middlebox
Packet In
Receive Packet Check For Criteria
Get Respective Packet Send Out
Packet Out

Figure 4.1: Illustration of the individual parts of the DPDK application for priority scheduling. The left part is responsible for receiving and distinguishing packets according to certain criteria. The right part sends out a packet according to the identified criteria and predefined weights.

The sending of the corresponding packet is done by removing the first packet from the FIFO or priority queue respectively and using the DPDK ring buffer to send. However, since there are two queues, each having different relevance, a scheduling procedure has to be deployed. For that, a weighted round-robin scheduling algorithm has been chosen to remove packets from the queues according to predefined weights, thus giving more importance to the traffic/queue having the higher weight. Figure 4.2 illustrates the above described procedure and shows the packet flow within the application.

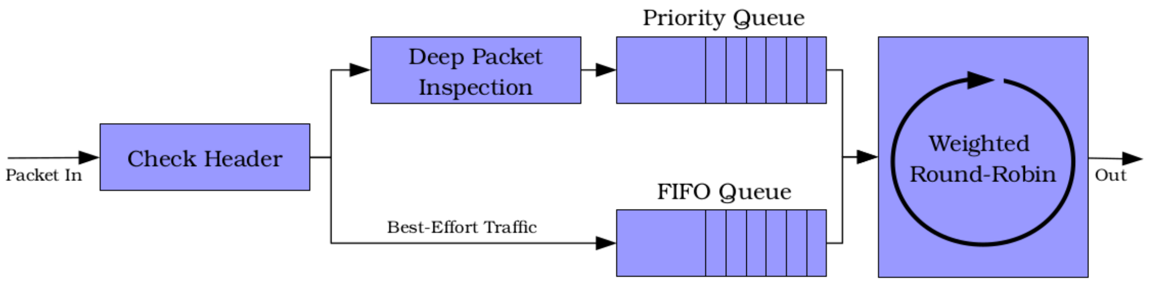


Figure 4.2: Illustration of the packet flow within the priority scheduling application. Whenever a packet is received, the header is checked for a specific characteristic. If the field matches a predefined criteria, the priority is extracted and the packet is enqueued into a priority queue, whereas a FIFO queue is used for non-matching best-effort traffic. Finally, the packets are sent out according to a predefined weighted round-robin procedure.

In chapter 6 this application is evaluated regarding the stability and performance of each individual NCS compared to standard round-robin scheduling application (see page 36).

4.2 In-network Priority Adaptation

An extension of priority scheduling using NFV is in-network priority adaptation. So far, priorities have been computed on end systems. However, end systems may not know about the state of the network due to the lack of a global view, which is why these priorities may not be used for routing decisions in case of congestions within the network. Using OpenFlow, these priorities can be adapted within the network on designated switches. Furthermore, priority-based routing could be deployed to route high priority packets on shorter paths while less important traffic can be routed on possibly longer ones. The scheduling of the high priority control traffic of individual Networked Control Systems can be done by configuring the switches to route packets having specific ranges of DSCP or a distinct VLAN PCP to one of eight different per-port queues. However, since DSCP or VLAN is part of the Ethernet or IPv4 packet respectively, the queueing scheme can only be applied for discrete priorities. Using SDN and OpenFlow, the individual adaptation and routing mechanisms can be pushed to the respective switch on the fly whenever network changes are detected by the SDN controller, thus providing higher flexibility.

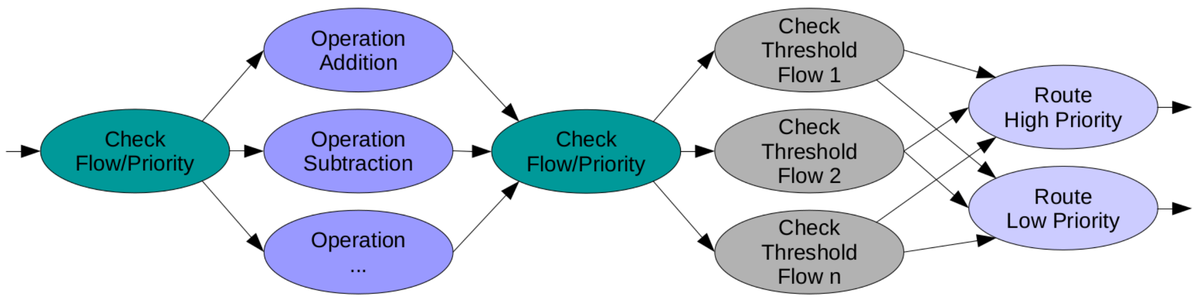


Figure 4.3: Illustration of individual flow tables and their connection using the priority adaption scheme. Depending on the flow, a different operation and threshold can be chosen, thus resulting in a different path (low priority vs. high priority path)

An important feature of OpenFlow is the pipelining of flow tables. With that, simple tables can be generated only responsible for single task instead of a single, giant table containing all possible combinations of match fields. Furthermore, different actions can be assigned to different flow tables, which allows for simple packet identification and handling.

With the use of the table pipelining feature, an OpenFlow controller as well as a library offering a simple interface to the controller were implemented for priority adaptation and scheduling (see https://github.com/ZnSn/Masterthesis.git). In Figure 4.3 the functionality and sequence of operations in a switch configured by this controller can be seen. The traversal sequence of each individual table for arriving packets is as follows:

1. Whenever a packet arrives at the switch the affiliation to the NCS is checked. This is done by matching the packet's header with predefined criteria such as source address and priority (VLAN PCP/DSCP).
2. Depending on the flow, the packet is forwarded to one of the installed tables used for increasing or decreasing the priority.
3. The next table's match field is an exact copy of the first one and used to distinguish the flows for the "Check Threshold" tables. Depending on the flow, a different threshold may be used as an indicator for the routing decision.
4. Whether the flow's priority matches the threshold or not is determining whether the packet is sent out on the low or high priority path.

The controller is implemented using the Ryu framework which implements the OpenFlow functions and offers a simple interface. Whenever a switch registers with the controller, the ID of the switch is compared to a predefined topology. If the ID matches a configuration of a switch defined in the topology, the configuration is applied to the switch using the flow modification messages provided by Ryu. These flow modification messages are used to push the flow tables reflecting the configuration to the switch. Since each individual table is used for a different task and the table numbering while traversing the pipeline, as defined by OpenFlow, has to be increasing, start at zero, and has a maximum of 255, the library offers predefined table numbers and ranges:

Table	Table Number
Check Flow/Priority	0
Operations	1–19
Check Flow/Priority	20
Check Threshold	21–253
Routing	254 & 255

Table 4.1: Table Numbering and Ranges

Hence, by using this table numbering convention, 19 operations and 232 individual flows are supported. However, by changing the respective definition in the library, more or less operations/flows could be provided. In detail, the individual flow tables are the following:

Check Flow/Priority:

This table is used to distinguish distinct flows from one another by using header fields like source address or DSCP. Depending on the flow, a go-to action with different destination table number is applied reflecting the desired operation for that particular flow.

table	priority	dl_vlan	dl_src	actions
0	1	1	00:00:00:00:00:01	goto_table:1
0	1	2	00:00:00:00:00:02	goto_table:2
0	1	3	00:00:00:00:00:03	goto_table:2
0	1	4	00:00:00:00:00:04	goto_table:20

Table 4.2: Sample flow table of flow/priority identification

Table 4.2 illustrates a sample flow table of flow/priority identification. Here, 4 different flows are registered, identified by dl_vlan (VLAN priority) and dl_src (source ether address). Depending on the flow, a different table is chosen as the next hop (i.e. addition, subtraction or skipping).

Operation X:

For each operation, there is a table representing the desired behavior. Unfortunately, OpenFlow does not provide actions such as "increase priority (VLAN PCP/DSCP) by one" so that other techniques have to be employed. Table 4.3 shows such a technique. There are table-entries matching the priority field (in this case VLAN) of each possible priority. Depending on the match, the action is set to first change the respective priority field to the desired value (in this case an increase by one) and to then continue with the next table.

Other possible operations include the reduction of the priority by a predefined value, a change of priority by a given function and the identity operation which does not change the priority at all and is implemented by directly forwarding the packet to the next table in the pipeline.

table	priority	dl_vlan	actions
1	1	0	set_field:1->vlan_pcp,goto_table:20
1	1	1	set_field:2->vlan_pcp,goto_table:20
1	1	2	set_field:3->vlan_pcp,goto_table:20
1	1	3	set_field:4->vlan_pcp,goto_table:20
1	1	4	set_field:5->vlan_pcp,goto_table:20
1	1	5	set_field:2->vlan_pcp,goto_table:20
1	1	6	set_field:7->vlan_pcp,goto_table:20
1	1	7	set_field:7->vlan_pcp,goto_table:20

Table 4.3: Sample flow table of the addition operation

Check Threshold Flow Y:

This table is used to decide, which routing path should be taken, high or low priority, depending on the priority of the packet and the respective threshold. Since each flow may have different demands regarding the threshold priority and routing, there is one table for each flow.

table	priority	dl_vlan	actions
20	1	0	goto_table:254
20	1	1	goto_table:254
20	1	2	goto_table:254
20	1	3	goto_table:254
20	1	4	goto_table:255
20	1	5	goto_table:255
20	1	6	goto_table:255
20	1	7	goto_table:255

Table 4.4: Flow-table for checking the threshold

When a packet is forwarded to the table, the priority (VLAN or DSCP) is checked. If this priority is higher than a predefined threshold given by the configuration, the packet is forwarded to the table responsible for high priority routing (and if not to the low priority routing table). Table 4.4 shows an example of such a table.

Route High/Low Priority:

These two tables are used for routing the packets on the high or low priority path respectively. Since for each NCS a different port and/or route could be chosen by the SDN-controller, the table's match-field includes the individual flow-identifier fields previously used in the "Check Flow/Priority" table.

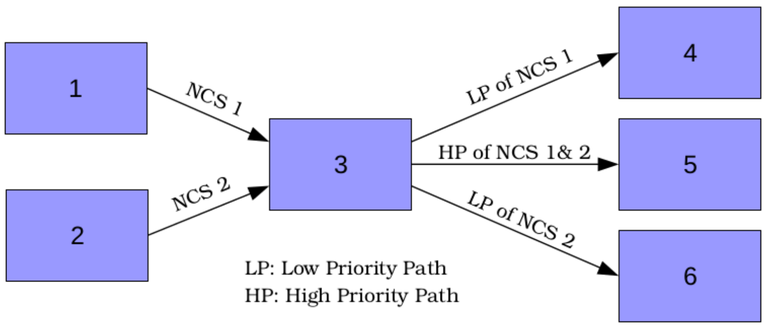


Figure 4.4: The routing of a switch supporting two different NCS. While the high priority path is the same for both NCS, the paths for low priority packets differ.

Figure 4.4 shows an example of a switch (3) routing two different flows of NCS. Whenever a high priority packet is received at the switch, it is forwarded on port 2 for both flows. For low priority packets, however, the behavior is different. While the flow with VLAN PCP of 1 is forwarded on port 1, the NCS having a VLAN PCP of 2 is forwarded on port 3. Table 4.5 shows the flow tables for this configuration.

table	priority	dl_vlan	dl_src	actions
255	1	1	00:00:00:00:00:01	output:2
255	1	2	00:00:00:00:00:02	output:2

table	priority	dl_vlan	dl_src	actions
254	1	1	00:00:00:00:00:01	output:1
254	1	2	00:00:00:00:00:02	output:3

Table 4.5: Low priority routing table supporting two flows corresponding to Figure 4.4. High priority packets are always sent out on port 2, low priority packets forwarded on port 1 and 3 for NCS 1 and NCS 2 respectively.

5 Networked Control System Simulation

For evaluating the applications described in the previous chapter, a working Networked Control System is needed. Since this work is not focusing on the control system but on the network, a simulation of the control system and emulation of the network is sufficient for the evaluation. For that reason, such a simulation has been implemented and is explained in detail in this chapter. For implementation details, see https://github.com/ZnSn/Masterthesis.git

5.1 Simulation Details

For designing and implementing a simulation of a Networked Control System, a mathematical model of the communication channel and the plant is required. In this thesis, the model by Carabelli et al. [10] on the basis of Blind and Allgöwer [25] has been implemented. In this model, a constant delay T_s and arrival probability p_a is assumed for the communication between the controller and a sensor. In addition, the model assumes an identically distributed and independent loss rate. The simulation uses the following differential equations for modeling the plant

(5.1) $\dot{x}(t) = Ax(t) + Bu(t) + w(t)$

(5.2) $y(t) = Cx(t) + v(t)$

where x represents a vector of the state of the plant (measured by the sensor), u is the input vector of the plant and w and v are modeling random disturbances such as noise and uncertainties. The input vector u is simultaneously the measurement signal, which is periodically transmitted over the network every T_s time units. Hence, this network delay T_s is at the same time the sampling period of the system. The controller is designed to solve the linear-quadratic regulation (LQR) problem, which uses a mathematical algorithm that minimizes a weighted cost function (For details see [26]).

In each sampling period, the simulation performs two actions: First, the sensor takes a measurement y of the system's current state x. This measurement is sent to the controller over the network at time t_k and is delivered at the time t_{k+1}. Second, depending on whether the controller received the previous measurement, the simulation either uses the received value to calculate a prediction $\hat{x}(t_k)$ of the current state or uses the previous state estimate $\hat{x}(t_{k-1})$ to calculate the current state estimate. The estimation of the state from received measurements is performed using a standard Kalman filter. The calculated value is then used to obtain the control input $u(t)$ by applying the following formula:

$$u(t) = -K\hat{x}(t_k), t \in (t_k, t_{k+1}] \tag{5.3}$$

where K represents the controller gain matrix used for the state prediction. For more details regarding prediction and calculations see [26]. The resulting control input is then used to update the internal state x_{t+1} for the next sampling period.

In this thesis, the details and models explained above are used to describe an inverted pendulum (see Figure 5.1). The pendulum is controlled by a cart, which can be accelerated and moved along the x-axis by the actuator. Depending on the pendulum's angle and angular velocity as well as cart position and acceleration, detected and monitored by the sensor, the cart has to be moved in the corresponding direction to keep the pendulum upright.

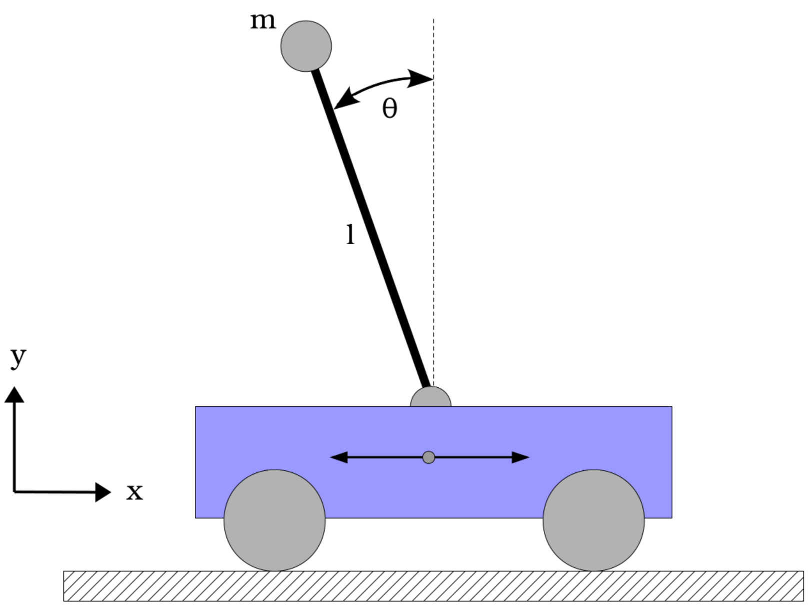


Figure 5.1: Illustration of an inverted pendulum.

5.2 Architecture

The implemented simulation is split into two parts: A library for control system specific calculations and a middlebox. While the library is responsible for handling all simulation/system internal calculations and state updates, the middlebox handles the control of the library and network.

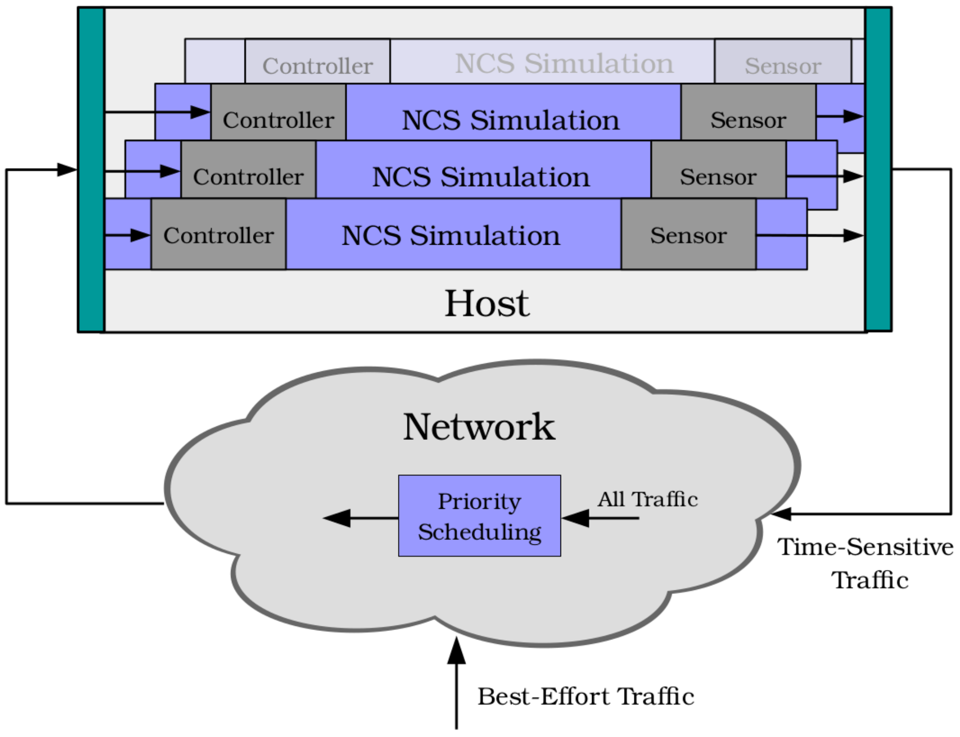


Figure 5.2: The architecture of the middlebox and network part of the NCS simulation. A single host is running one or multiple instances of NCS. All instances are handled by this middlebox and are using the same interfaces.

The middlebox is implemented using DPDK and is, as can be seen in Figure 5.2, running one or multiple instances of NCS simulations. These simulations are all connected to the same interface and are handled by the DPDK application. Each instance is split into sender (sensor) and receiver (controller + actuator). The individual sender's task is to start the state calculation, send the respective message over the network and trigger the controller. The corresponding controller's task, on the other hand, is to retrieve the correct packet from a queue and initiate system internal calculations.

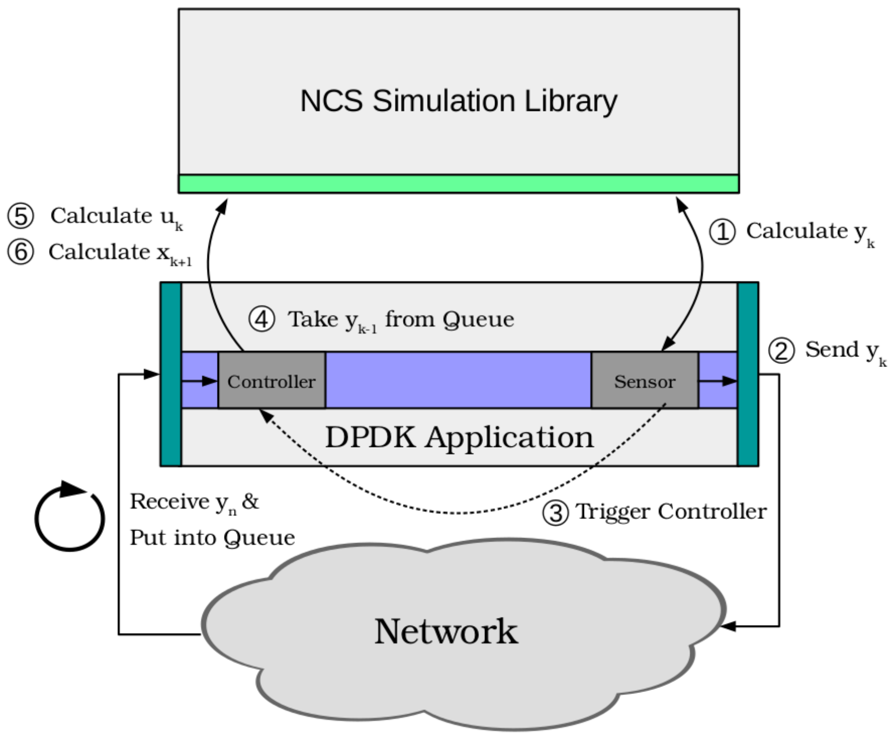


Figure 5.3: Structure and interaction of the NCS simulation and the application.

Figure 5.3 illustrates the complete structure and shows the interaction between library and application. In detail, the sequence of operations is as follows:

1. **Calculate current $\mathbf{y}_k$:** The sensor calculates an output measurement (usually a subset of the plant's internal state variable x) for further control calculations.
2. **Send calculated $\mathbf{y}_k$:** The calculated value is sent over the network to the controller. At the receiver, it is added to a queue instead of being processed immediately, allowing for synchronization between sensor and controller part of the application.
3. **Trigger controller:** For synchronization purposes, the controller is triggered after the sensor has finished its state update calculations and the updated values were sent over the network.
4. **Retrieve correct $\mathbf{y}_{k-1}$:** Since the controller is triggered immediately after the sensor has sent the update, the controller's simulation time (k-1) is the one before the sender's (k) so that y_{k-1} has to be taken from the queue for further calculations.
5. **Calculate $\mathbf{u}_k$:** Using y and the internal state, the control variable u is calculated.
6. **Calculate and update $\mathbf{x}_{k+1}$:** The new control variable u is now used to update the internal state for the next sampling time.

6 Evaluation

In this chapter, the individual contributions are evaluated. At first, the performance of the implemented NCS simulation is examined and a comparison regarding stability and performance between a connection using DPDK and sockets is made. Later, the implemented priority scheduling application using DPDK is evaluated with respect to the number of NCS that could be supported compared to a standard round robin procedure. Finally, the OpenFlow application is examined and the impact of the pipeline is measured.

6.1 NCS Simulation

The NCS simulation is a crucial part for evaluating the developed applications. In the following, the differences between a simulation running on sockets compared to a simulation running with DPDK is explored. In addition stability measurements are taken with respect to the configuration and control parameters (e.g. delay, arrival probability, losses)

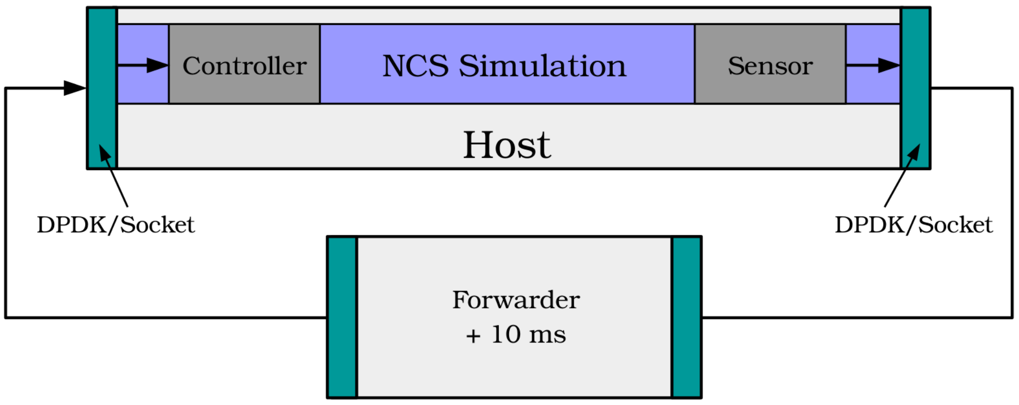


Figure 6.1: Evaluation setup for the NCS simulation. The sensor part of the simulation is connected to a simple host responsible for delaying the packet for 10 ms to simulate a constant network delay.

Figure 6.1 is illustrating the evaluation setup. Only one NCS simulating an inverted pendulum is run on a Host connected to a middlebox. The middlebox is running a simple DPDK application responsible for forwarding the incoming traffic to the controller with a 10 milliseconds delay. This delay is added since the time it takes the simulation to do the necessary calculations is longer than the time it would take the packet to be transmitted via a direct link, which would result in the minimum possible sampling time to be longer than the time necessary to do the calculations. By using the delay, the sampling time of the system can now be chosen such that it is close to the network delay. This gives the opportunity to evaluate the behavior and the performance of the system with respect to packets arriving to late at the controller.

6.1.1 Simulation Performance

To measure the performance of the NCS simulation, two system parameters, the pendulum's angle and the performance factor J, are considered. While the angle is included in the systems internal state, J has to be computed using the state x. In each sampling time, the current system performance J_{abs} is calculated using the following formula:

(6.1) $J_{abs}(t) = x(t)' * Q * x(t) + 2 * x(t)' * H * u(t) + u(t)' * r * u$

At the end of the simulation, the overall performance J of the system is computed by averaging the intermediate values of the system's performance and dividing it with the sampling time T_s.

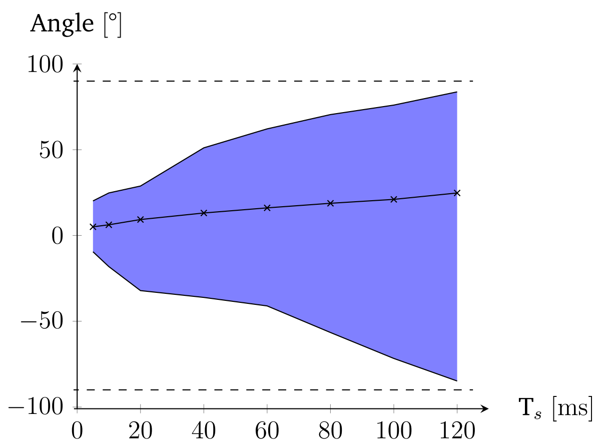
(6.2) $J = \frac{\sum_{i=0}^{n} \frac{J_{abs}(i)}{n}}{T_s}$

For evaluation using these two parameters, the middlebox was removed and replaced by a direct link. The system is considered to be unstable and stopped when the angle of the pendulum exceeds 90°. Each simulation is executed for 300 iterations.

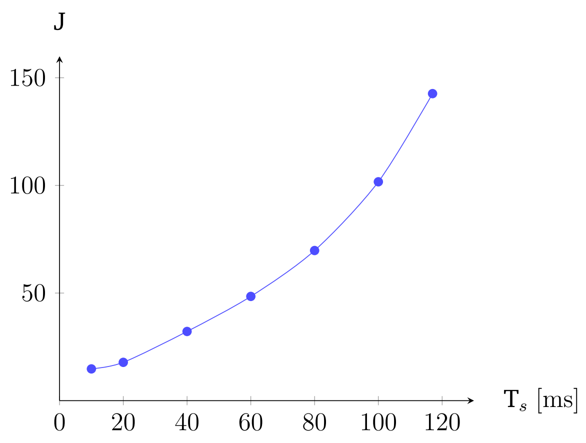
T_s [ms]	10	20	40	60	80	100	120	130
Maximum	0.433	0.503	0.891	1.084	1.230	1.326	1.461	–
Minimum	-0.316	-0.561	-0.630	-0.716	-0.987	-1.251	-1.480	–
Average	0.109	0.162	0.229	0.281	0.327	0.367	0.432	–

Table 6.1: Minimal/maximal angle (radian) and average of absolute angles.

Table 6.1 shows the minimal and maximal angle in radian of the pendulum as well as the average of absolute angles over the complete simulation time with different sampling times. The maximum possible sampling time of the NCS was found to be 127 ms. Furthermore, as can be seen in Table 6.1, the average absolute angle is increasing with the sampling time. This is due to the fact that the higher the sampling time, the later the controller receives the state measurement and the later the system can react to the pendulum falling over.



(a) Range of Pendulum Angles



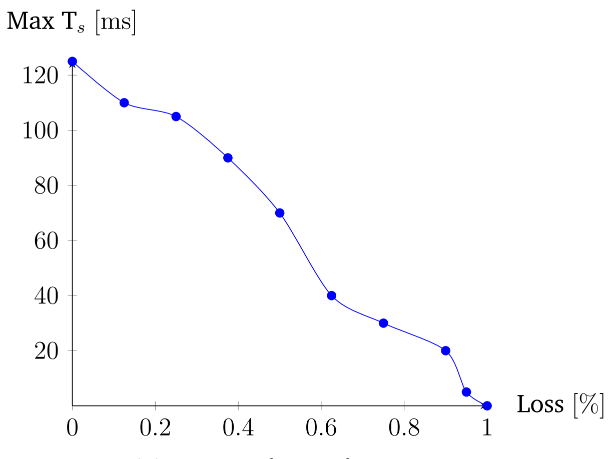
(b) Performance Factor J

Figure 6.2: The range of the pendulum angles as well as the average absolute angle of the pendulum (6.2a) on the left and the performance J of the system (6.2b) on the right depending on the sampling frequency T_s. The dashed lines in the left figure represent the stable area of the system (-90° to +90°).

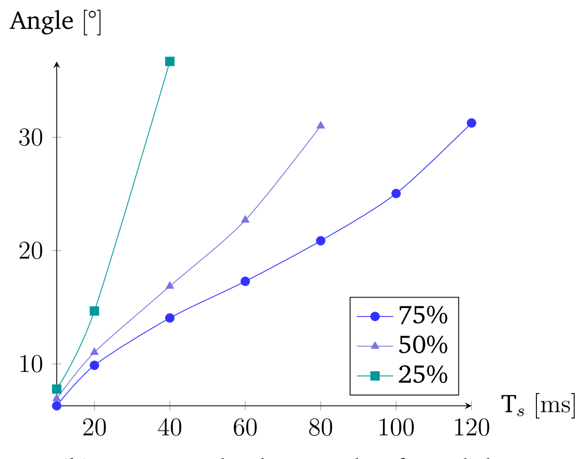
The same applies to the minimal and maximal angles of the pendulum. Since the system is slower to react to changes in the pendulum's angle, the pendulum has more time falling over, which in turn leads to higher minimal and maximal angles. When looking at the performance J, it can be seen that the value also increases with the sampling time. Since J is using the internal state of the system for calculation and the internal state consists, among other things, of the angle and the angular velocity (in the case of the pendulum), it is reasonable that this parameter also increases when the angle is high and the system is becoming more and more unstable (see Figure 6.2).

6.1.2 Stability Measurement

In the following, the stability of the system with respect to packet losses is measured. Each outgoing packet is sent out with a probability of 75, 50, or 25 percent or not sent at all otherwise. As could be seen in the previous section, the simulation running without loosing any packets was stable until a sampling frequency over 120 milliseconds was reached. Figure 6.3a shows the maximal possible sampling times with respect to the loss probability.



(a) Maximal sampling time T_s



(b) Average Absolute angle of pendulum

Figure 6.3: Average absolute angle of the pendulum depending on the sampling time T_s and the loss probability (6.3a) on the left and the average absolute angle of the pendulum depending on the sampling time and the loss probability (6.3b) on the right.

The higher the loss probability, the more control packets are missed by the controller. Since each missed packet limits the controllers ability to stabilize the pendulum, the sampling time has to be decreased in order to compensate the losses and inaccuracies in the prediction of the control system. The same statement holds for the average absolute angle of the pendulum. The more packets are lost, the faster the average angle increases and the faster the angle reaches its maximum of 90°.

6.1.3 Comparison DPDK vs. Sockets

This section focuses on the performance and stability of the simulation on the system's boundaries (e.g. close to the minimal possible sampling time) and compares the usage of DPDK to a connection using standard sockets. However, when using a direct link, as was done in the previous sections, the simulation time is higher than the system's minimal sampling time. For that reason, all traffic is routed through a middlebox, which delays the incoming packet by 10 milliseconds and thus inducing an artificial network delay.

Since the sampling time is now chosen so that it is close to the minimal possible sampling time (e.g. 10 ms network delay plus the time the simulation takes for calculations, sending, and receiving), the jitter is an important factor when it comes to the stability of the system. The higher the jitter, the more packets could arrive too late at the controller, thus being futile for the current calculations.

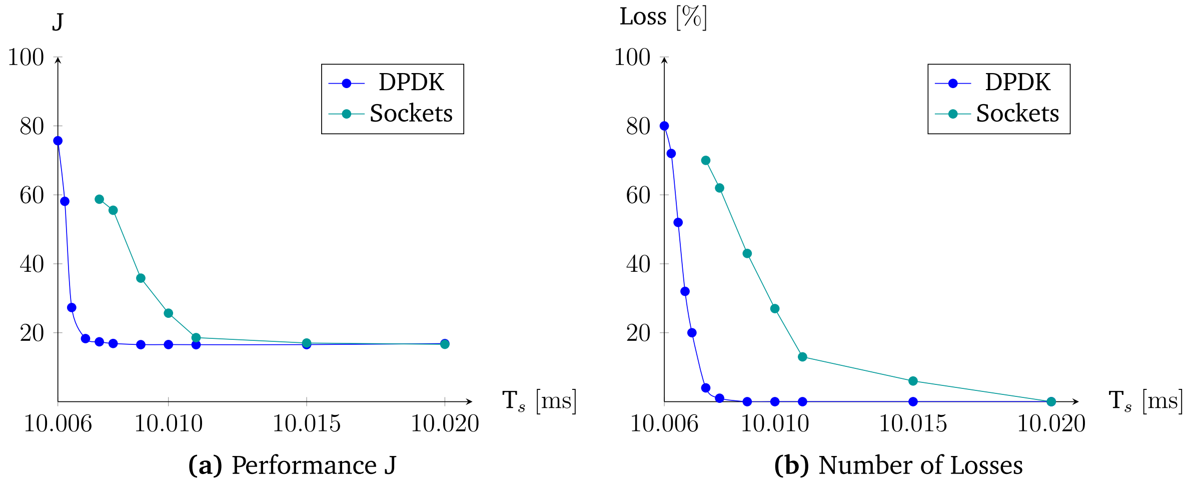


Figure 6.4: The performance factor J (6.4a) on the left and the number of losses (6.4b) on the right for different sampling times of the DPDK application compared to the connection using sockets.

In the experiments, the sampling time was changed to range from 10.006 milliseconds as the minimal value to 10.020 milliseconds. For each experiment, the performance factor J and the overall percentage of lost packets were used as key factors for evaluation. Figure 6.4 shows the results for both J and the loss rate.

As can be seen, the system using DPDK is able to stabilize the pendulum for smaller sampling times than the system using sockets. While sockets were not able to go beneath a sampling period of 10.0075 milliseconds, DPDK became unstable starting at 10.006 ms. Another important key point to consider is that the socket application is starting to loose more packets at a higher sampling frequency than DPDK, which is also reflected in the performance factor J. This behavior, earlier instability and performance/packet losses, can be traced back to the increased amount of jitter and delay when using sockets compared to DPDK. While DPDK is bypassing the Kernel by using designated drivers and a ringbuffer for transmission and reception of packets, sockets are connected to the operating system's network driver, thus imposing additional delay and jitter by copying the data through the individual layers.

6.2 Priority Scheduling Application

For evaluation of the priority scheduling application, two of the ports of the middlebox are connected to the NCS simulation as well as a best-effort traffic generator via 10 Gbps links. All traffic is then scheduled and sent out over a third port. However, since in this setup the port used for sending is directly connected to the host running the simulation instead of a real network, all best-effort traffic is dropped upon arriving at the host. Figure 6.5 illustrates the setup and its connections.

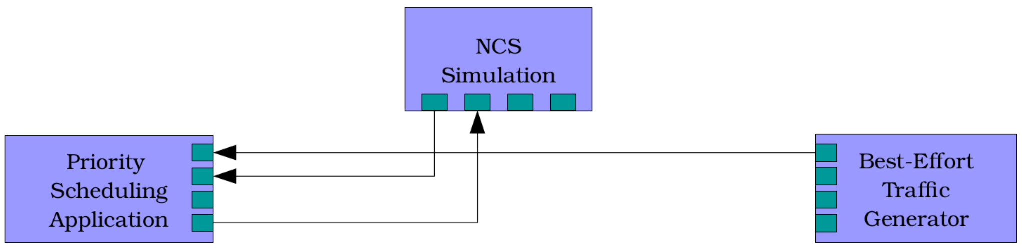
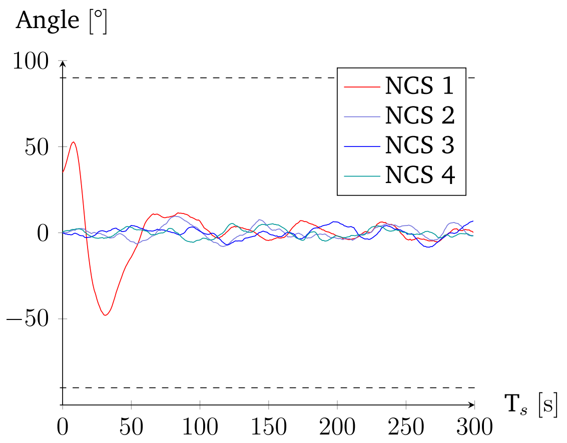


Figure 6.5: Setup for the evaluation of the priority scheduling application. The application is connected to the NCS Simulation as well as to a best-effort-traffic generator. All traffic is sent out over the same physical link.

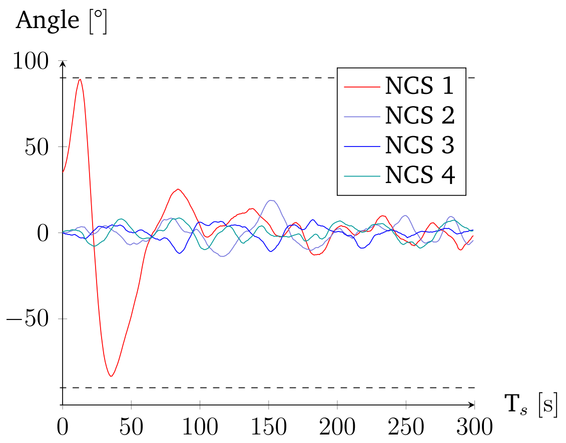
In the following section, the priority scheduling application is compared to a standard round-robin procedure regarding the overall stability of the systems with respect to the pendulums angle. The setup is as follows:

- The network is shared between 6 NCS.
- All NCS are running on the same DPDK host.

- NCS 1 is started with a pendulum having an initial angle of 35 °.
- To simulate a lower bandwidth, the middlebox generates tokens for the sending.
- Only if the sender has available tokens, a priority packet can be sent out.
- The queue size is set to 2. The queue is emptied after each sampling time.
- The sampling time T_s of the simulations is 50 milliseconds. In each sampling time, 2 tokens are generated.



(a) Priority Scheduling



(b) Round-Robin

Figure 6.6: Results of the experiment comparing the pendulum's angles of priority scheduling (6.6a) to round-robin (6.6b). For the experiment, 6 NCS were used, all running through the middlebox (although only 4 are shown) and using a sampling time of 50 ms. The queue size was set to 2.

Figure 6.6 shows the result of the experiment. Since NCS 2–6 all have the same initial parameters, only NCS 2–4 are shown in the following results. As can be seen, the fluctuation of the pendulum's angle when using the round-robin procedure is much higher than the one using priority scheduling. This is due to the fact that each NCS always misses, in this example, 2 sampling periods before receiving a control message when using round-robin. The priority scheduling application, on the other hand, uses the priority assigned to the packet by the control system's sensor, which in turn is alloted depending on the internal and predicted state of the respective system. This means that more unstable systems will calculate a higher priority value, which will then be sent out before a more stable systems.

	NCS 1		**NCS 2**		**NCS 3**		**NCS 4**	
	RR	Prio	RR	Prio	RR	Prio	RR	Prio
$\bar{x}$	0.121	0.011	0.185	-0.023	-0.054	-0.066	-0.226	-0.009
σ^2	472.73	147.744	31.176	12.647	26.014	10.632	19.162	9.327
σ	21.742	12.155	5.584	3.556	5.100	3.261	4.377	3.054
J	47.987	13.678	5.279	2.066	4.340	1.827	3.752	1.767

Table 6.2: Mean, variance, standard deviation and performance factor J of priority scheduling compared to round-robin.

In Table 6.2, the mean, variance and standard deviation of the pendulum's angle as well as the performance factor J of each individual NCS is shown for both priority scheduling and round-robin. As also reflected in Figure 6.6, the variance and standard deviation of the round-robin procedure is significantly higher than the values achieved with priority scheduling, which can also be attributed to the higher fluctuations and periodically missed packets. This characteristic is also responsible for the higher performance value (with the lower the better) of the round-robin procedure. As previously described in section 6.1, a bigger angle of the pendulum can be linked to a higher value of J.

In addition, as can be seen in both Table 6.2 and Figure 6.6, the values of the pendulum of NCS 1 are significantly higher than the respective counterparts of the remaining NCS. This characteristic comes from the initial internal state of NCS 1, which has been set to start with an angle of 35°, thus being less stable than the remaining NCS. In both cases, priority scheduling and round-robin, the controller needs time to stabilize the pendulum. When directly comparing the angles of the pendulum, Figure 6.6 clearly shows that, using the priority scheduling procedure, the pendulum reaches a maximum angle of about 52°. In comparison, the round-robin approach results in the pendulum almost reaching 90°.

Further experiments were conducted where the sampling time was varied to range from 50–125 milliseconds. Since the angle of the pendulum of NCS 1 has been chosen to be 35° instead of 0, NCS 1 became unstable starting at a sampling period of 50 ms while using the round-robin procedure. By contrast, the priority scheduling application was able to handle a sampling time of up to 120 milliseconds, which is almost the maximum possible sampling time (=127ms) of one individual NCS connected via a direct link (see subsection 6.1.1: Simulation Performance).

For additional comparison, the initial state of the pendulum has been changed to [0.4297, 1.9253, -0.9903, 0.5806]. This state is reflecting a pendulum with an angle of 56° (0.9903 in Radian), an angular velocity of 0.58$\mathrm{m\,s^{-1}}$, a cart x position of 0.4291 meter, and a cart acceleration of 1.9253 $\mathrm{m\,s^{-2}}$. Running the same experiments as above, NCS 1 became unstable starting at a sampling time of 25 milliseconds using round-robin, whereas sampling times of 120 were possible using priority scheduling. This clearly shows that, even if the pendulum is initialized with more unstable settings, priority scheduling is still able to keep the system stable while round-robin is failing at a much lower sampling time.

6.3 OpenFlow Library

The OpenFlow library and the associated controller are used to schedule incoming packets on an OpenFlow switch as well as to adapt the priorities according to predefined criteria. In the following, the time it takes to install a complete pipeline on the switch is measured on a physical switch. Furthermore, the impact of multiple NCS registered on the switch compared to the size of the pipeline in terms of the number of table-entries is evaluated.

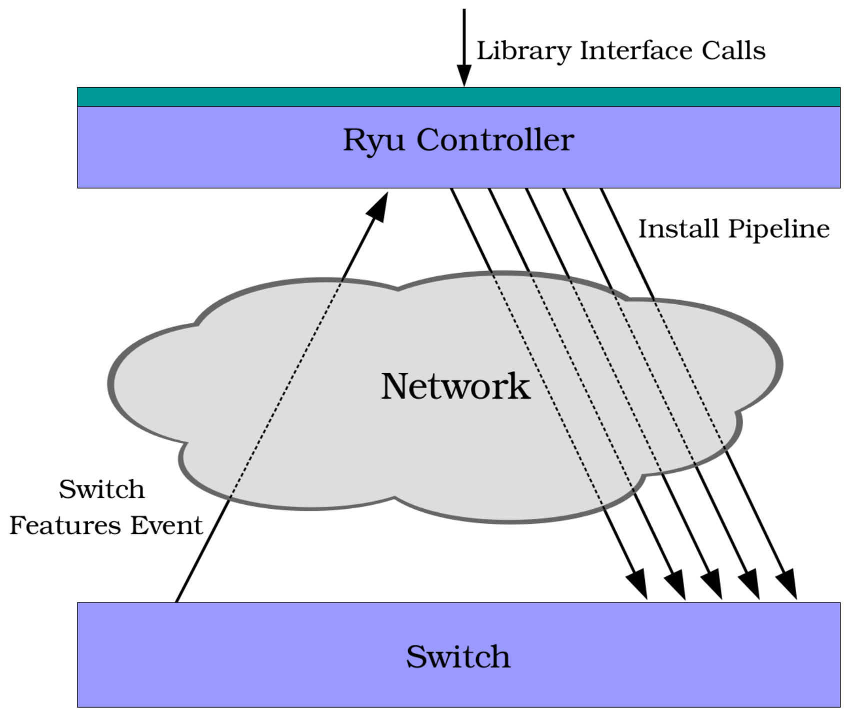


Figure 6.7: Setup for the evaluation of the OpenFlow module. The application using the library is configuring the controller. Whenever a switch registers with the Ryu controller, the complete pipeline is installed.

The evaluation setup can be seen in Figure 6.7. The application is using the library's interface to configure the controller to support a predefined topology. The controller is then using specific events triggered by a registering switch to install the pipeline with the specified flow tables on that switch. For the following evaluations, the switch and the controller are connected via a direct link.

6.3.1 Installing The Pipeline

The procedure for in-network priority adaptation described in section 4.2 is meant to be installed on the OpenFlow switch on the fly. For that to work, the installation time of the pipeline has to be short enough for the controller to react to network changes in time. This section discusses the amount of time it takes to install a complete pipeline on the OpenFlow switch with respect to the number of registered NCS within the network.

Since the flow table commands sent by the controller are non-blocking non-returning functions, a mechanism to measure the time for operations on the switch is needed. For that, a barrier message has been used in order to catch the completion of the installation process. Immediately after all commands are sent to the switch, a barrier message is sent to the same switch. This barrier message will trigger a reply from the switch when there are no more pending flow-modifications/switch-instructions present. This reply can be used to measure the time the complete pipeline installation process took.

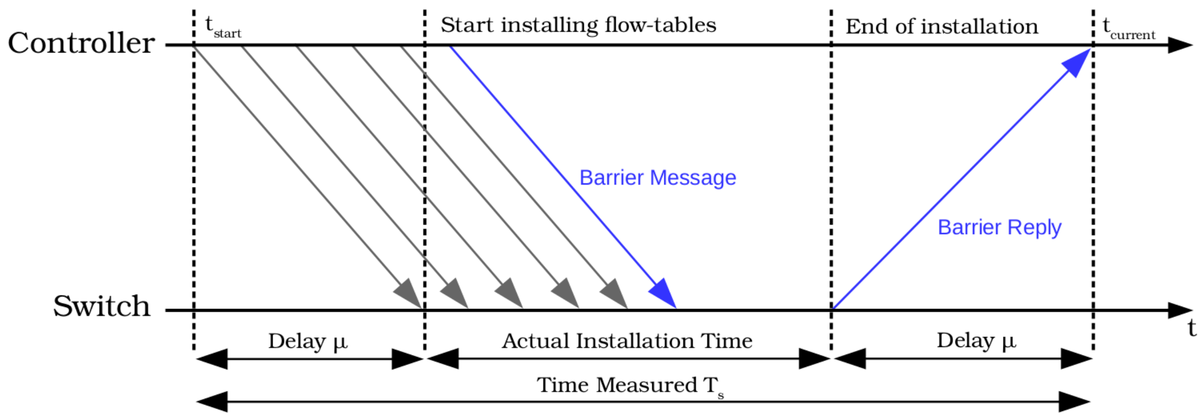


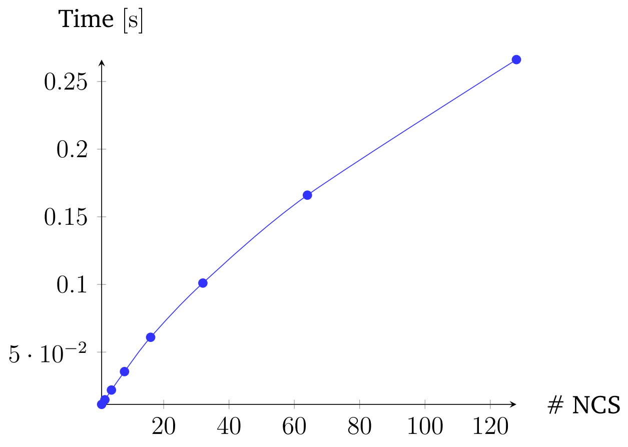
Figure 6.8: A barrier message is used to capture the time to install a complete pipeline on the switch. Following the last flow modification message, a barrier message is sent to the switch. When the switch completes all pending operations, a barrier reply message is sent back to the controller. The actual time is the measured time minus two times the network delay.

Figure 6.8 illustrates this procedure. Whenever a registered switch connects to the controller, the following steps are taken to measure the time required for the installation of a complete pipeline:

- **Start a timer:** The current time t_{start} is retrieved and used as the starting point of the installation process.
- **Send pipeline messages:** The controller starts sending the individual flow-modification messages.
- **Send barrier message:** After the pipeline messages were sent, a barrier message is transmitted to capture the completion of table modifications.
- **Switch sends barrier reply:** Once all flow-modification messages were executed and there are no pending operations, the switch sends a barrier reply message to the controller.
- **Calculate time:** The controller receives the barrier reply message via an event and starts calculating the installation time. This is done by retrieving the current time $t_{current}$ and calculating the time with

$$T_{install} = t_{current} - t_{start} - 2 * \mu \tag{6.3}$$

The results can be seen in Figure 6.3 and the table next to it. Since the controller and switch are connected via a direct link, the network delay is insignificant compared to the installation time such that the following results include that delay.



Number of NCS	Time [s]
1	0.0113
2	0.0147
4	0.0219
8	0.0355
16	0.0609
32	0.1010
64	0.1659
128	0.2661

Figure 6.9: Effect of the number of NCS on the time the pipeline takes to install.

As can be seen, the time it takes to install the pipeline is increasing with the number of flows within the network. This is due to the fact that the switch has to install more flow table entries the more NCS in the system. While only one NCS/flow takes 11.3 milliseconds, installing the pipeline for 128 NCS takes about 266.1ms.

6.3.2 Table Magnitude

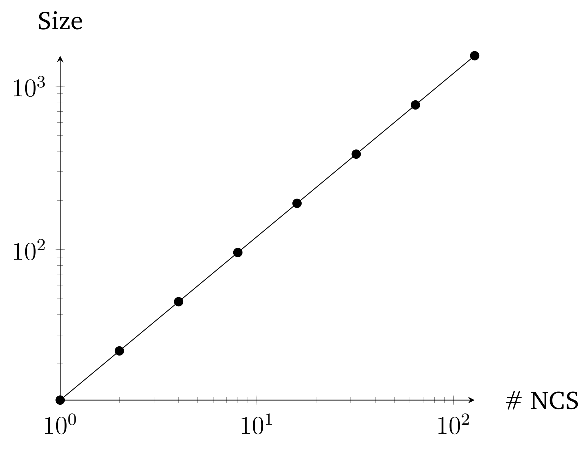
This section focuses on the size of the individual tables with increasing number of NCS. As can be seen in Figure 4.3 in chapter 4, the tables responsible for the adaptation of the priority (i.e. increase/decrease) are independent from the number of configured flow/NCS since they are just checking the packet's priority and adapt it according to the operation. All remaining tables, however, rely on the number of NCS since they add table-entries with each additional flow/NCS.

The table "Check Flow/Priority" (see Figure 4.3) checks for header fields (i.e. source address, DSCP, VLAN, etc.) to identify each individual flow. Hence, with increasing number of NCS, more entries reflecting that particular flow have to be inserted in this table. In addition, there is a "Check Threshold" table determining the respective routing table. Since there is a table for each individual flow, the number of tables increase with each NCS. Furthermore, the two routing tables, too, are depending on the number of flows since they have to forward the packets of the individual NCS according to the controller-defined path. In the following, the number of table-entries are metered with respect to the number of flows. Figure 6.10a and Table 6.4 show the results.

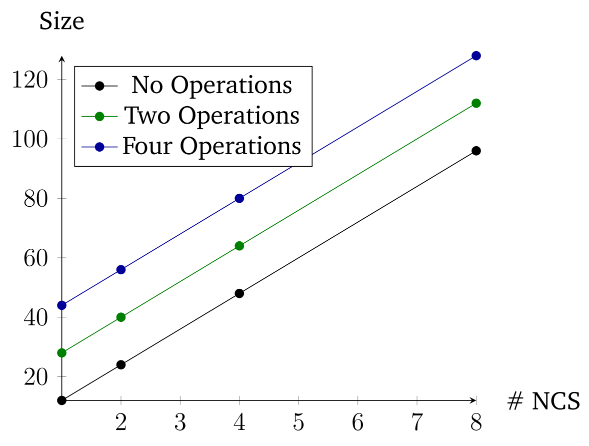
# NCS	Number of Table Entries					
	CF1	**Operations**	**CF2**	**Threshold**	**Routing**	**Sum**
1	1	$k * 8$	1	8	2	$12 + k * 8$
2	2	$k * 8$	2	16	4	$24 + k * 8$
4	4	$k * 8$	4	32	8	$48 + k * 8$
16	16	$k * 8$	16	128	32	$192 + k * 8$
128	128	$k * 8$	128	1024	256	$1536 + k * 8$

Table 6.4: Magnitude of the pipeline showing the number of flow-table entries depending on the number of configured NCS. The check Flow/Priority (CF) tables increase linear with the number of flows. The Operation tables have a constant number of entries only depending on the number (k) of installed operations. The threshold tables are installing 8 entries per flow.

As can be seen in Table 6.4, the number of entries in each of the previously mentioned affected tables increase with the number of NCS in the network. Since there is one flow-table entry for each NCS in the tables responsible for identifying the packet's affiliation, these tables have the same size as the number of NCS. The tables checking for the threshold of the packets priority for the routing decision have the highest impact on the size of the pipeline. Each flow has its own threshold table having eight entries (one for each possible priority). Therefore, the number of entries increase with 8 times the number of flows/NCS in the network.



(a) Pipeline Magnitude



(b) Operations Table Magnitude

Figure 6.10: Effect of the number of NCS within the network (6.10a) and the number of registered operations (6.10b) on the size of the pipeline.

All together, as shown in Figure 6.10, the size of the complete pipeline increases linearly with the number of flows. This also applies when considering the tables for priority adaptation. As shown, the impact of these tables is a constant one, thus maintaining the linear increase with the number of NCS.

6.4 Summary

To sum up, the results of the evaluation are the following:

- DPDK is able to achieve lower sampling times compared to sockets.
- The simulation running on sockets started missing packets earlier, thus achieving inferior performance compared to DPDK.

- The priority scheduling application is able to keep multiple systems stable more efficiently than a round-robin procedure.
- The priority scheduling application is able to stabilize a more unstable system faster, even for higher sampling times.
- The time it takes to install the pipeline on a switch and the space overhead increases with the number of NCS within the system.

7 Conclusion

This thesis presented approaches to create a global view on a network shared by multiple Networked Control Systems, where each of them require strict guarantees with respect to timeliness, delay, and jitter, along with applications requiring best-effort service only.

At first, a middlebox for priority scheduling on the basis of NFV was developed using DPDK. This middlebox splits best-effort from time-sensitive control traffic and orders the control-traffic according to continuous priorities given by the individual NCS in the packet's payload. Both kinds of traffic, best-effort and high-priority, are sent out using a standard token-based round-robin scheme, thus giving the opportunity to assign more or less importance to the best-effort or control traffic respectively.

In addition, a priority adaptation application was developed using Software Defined Networking and OpenFlow. This application is able to schedule traffic by using specific header fields of the packet as discrete priorities and mapping them to the internal per-port queues of a switch. Furthermore, the priority can be adapted according to predefined functions and forwarded on one of two different paths by configuring the switches flow tables. Using this approach, priority-based routing can be applied, thus giving more flexibility.

For evaluation, a simulation of a NCS was implemented, which simulates an inverted pendulum. Using this simulation along with an emulation of a network, the priority scheduling could be evaluated by comparing it to a standard round-robin procedure. Taken together, the priority scheduling application was able to achieve better stability, both in terms of the pendulums angle and also regarding the performance of the system. Additionally, a much higher sampling time of the individual systems was possible using priority scheduling (120ms) compared to round-robin (50ms). The adaptation scheme has been evaluated in terms of the time required to configure a switch and the number of tables-entries with respect to the number of NCS within the network.

Future Work

The approach using SDN and OpenFlow, as implemented in this thesis, is just providing the primitives in order to schedule and/or adapt traffic according to predefined header fields. However, the priorities in this thesis are considered to be preassigned by a higher level application. One topic for future research could be the development of a method determining how to assign these priorities based on the status of the network, the stability of each individual NCS and the number of NCS within the network.

Furthermore, OpenFlow only provides a matching on the packet's header fields. If, however, DPI is supported by OpenFlow in future releases, the implemented middlebox could be converted to run on a switch. In this case, the approach using SDN and OpenFlow could also be used to support continuous priorities embedded into the packet's payload, which in turn would open up the possibility to combine both approaches implemented in this thesis.

Bibliography

[1] T. C. Yang. “Networked control system: a brief survey.” In: *IEE Proceedings Control Theory and Applications* 153.4 (2006), p. 403 (cit. on p. 1).

[2] Y. Tipsuwan, M.-Y. Chow. “Control methodologies in networked control systems.” In: *Control engineering practice* 11.10 (2003), pp. 1099–1111 (cit. on p. 1).

[3] H. Voit et al. “An Arbitrated Networked Control Systems Approach to Cyber-Physical Systems.” PhD thesis. Technische Universität München, Diss., 2013, 2013 (cit. on pp. 1, 5).

[4] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig. “Software-defined networking: A comprehensive survey.” In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76 (cit. on pp. 2, 8, 10).

[5] H. Kim, N. Feamster. “Improving network management with software defined networking.” In: *IEEE Communications Magazine* 51.2 (2013), pp. 114–119 (cit. on pp. 2, 8, 10).

[6] L. Zhang, Y. Shi, T. Chen, B. Huang. “A new method for stabilization of networked control systems with random delays.” In: *IEEE Transactions on automatic control* 50.8 (2005), pp. 1177–1181 (cit. on p. 5).

[7] G. C. Walsh, H. Ye, L. G. Bushnell. “Stability analysis of networked control systems.” In: *IEEE transactions on control systems technology* 10.3 (2002), pp. 438–446 (cit. on pp. 5, 7).

[8] R. S. Raji. “Smart networks for control.” In: *IEEE spectrum* 31.6 (1994), pp. 49–55 (cit. on p. 5).

[9] J. P. Hespanha, P. Naghshtabrizi, Y. Xu. “A survey of recent results in networked control systems.” In: *PROCEEDINGS-IEEE* 95.1 (2007), p. 138 (cit. on pp. 5, 7).

[10] B. W. Carabelli, F. Dürr, B. Koldehofe, K. Rothermel. *A Network Abstraction for Control Systems*. Technical Report Computer Science 2014/01. University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems: University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2014, p. 25 (cit. on pp. 6, 7, 27).

[11] W. Zhang, M. S. Branicky, S. M. Phillips. “Stability of networked control systems.” In: *IEEE Control Systems* 21.1 (2001), pp. 84–99 (cit. on p. 7).

[12] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti. “A survey of software-defined networking: Past, present, and future of programmable networks.” In: *IEEE Communications Surveys & Tutorials* 16.3 (2014), pp. 1617–1634 (cit. on pp. 8, 10).

[13] M.-K. Shin, K.-H. Nam, H.-J. Kim. “Software-defined networking (SDN): A reference architecture and open APIs.” In: *2012 International Conference on ICT Convergence (ICTC)*. IEEE. 2012, pp. 360–361 (cit. on p. 8).

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner. “OpenFlow: enabling innovation in campus networks.” In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74 (cit. on p. 10).

[15] *OpenFlow Switch Specification*. Version 1.4.0. Open Networking Foundation. Oct. 2013 (cit. on pp. 10, 11).

[16] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba. “Network function virtualization: State-of-the-art and research challenges.” In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 236–262 (cit. on pp. 12, 13).

[17] B. Han, V. Gopalakrishnan, L. Ji, S. Lee. “Network function virtualization: Challenges and opportunities for innovations.” In: *IEEE Communications Magazine* 53.2 (2015), pp. 90–97 (cit. on pp. 12, 13).

[18] G. Wang, T. E. Ng. “The impact of virtualization on network performance of amazon ec2 data center.” In: *INFOCOM, 2010 Proceedings IEEE*. IEEE. 2010, pp. 1–9 (cit. on p. 13).

[19] N. G. Nayak, F. Dürr, K. Rothermel. *Time-sensitive Software-defined Networks for Real-time Applications*. Technical Report Computer Science 2016/03. University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems: University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2016, p. 24 (cit. on p. 14).

[20] *DPDK Documentation*. http://dpdk.org/doc. Accessed: 2016-07-05 (cit. on p. 13).

[21] L. Rizzo. “netmap: A Novel Framework for Fast Packet I/O.” In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 101–112. ISBN: 978-931971-93-5 (cit. on p. 14).

[22] B. Cho, M. K. Aguilera. “Surviving congestion in geo-distributed storage systems.” In: 2012, pp. 439–451 (cit. on p. 16).

[23] J. A. Perez, V. H. Zarate, C Cabrera, J. Janecek. “A network and data link infrastructure design to improve QoS for real time collaborative systems.” In: *Advanced Int’l Conference on Telecommunications and Int’l Conference on Internet and Web Applications and Services (AICT-ICIW’06)*. IEEE. 2006, pp. 19–19 (cit. on p. 16).

[24] C. Rodrigues, Y. Krishnamurthy, I. Pyarali, P. Gore. “Using Prioritized Network Traffic to Achieve End-to-End Predictability.” In: *Real-Time and Embedded Distributed Object Computing* (2002), pp. 15–18 (cit. on p. 16).

[25] R. Blind, F. Allgöwer. “Is it worth to retransmit lost packets in Networked Control Systems?” In: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. IEEE. 2012, pp. 1368–1373 (cit. on p. 27).

[26] E. D. Sontag. *Mathematical Control Theory: deterministic finite dimensional systems*. Vol. 6. Springer Sicence & Business Media, 2013 (cit. on pp. 27, 28).

Declaration