Predicting Traffic Flows for Traffic Engineering in Software-Defined Networks

Master-Thesis von Emmanuel Stapf
Oktober 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Predicting Traffic Flows for Traffic Engineering in Software-Defined Networks

Vorgelegte Master-Thesis von Emmanuel Stapf

1. Gutachten: Prof. Dr. techn. Gerhard Neumann
2. Gutachten: Patrick Jahnke

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 4. Oktober 2016

(Emmanuel Stapf)

Abstract

Traffic congestion is a well-known problem in networks that inevitably leads to packet loss and a reduction of the network throughput, which in turn lowers the quality of the services running in the network. The recently emerged concept of Software-Defined Networking offers unique features for implementing powerful new traffic-engineering systems to cope with the problem of network congestion. The advantages of Software-Defined Networks (SDNs) compared to traditional networks arise from the presence of a central entity in the network that can obtain a global view on the current network situation and programmatically change the routing behavior of the network devices in real time. Nevertheless, current traffic-engineering systems developed for SDNs cannot sufficiently prevent congestion because they only react on the current network situation. The ability to predict the network traffic to a certain degree would allow a traffic-engineering system to reroute the traffic before the congestion situation occurs. In previous research on network traffic prediction, real traffic data was processed by combining many single flows and by aggregating the flows on a temporal scale. The made predictions are inapplicable for traffic-engineering systems because they do not allow for making rerouting decisions on a per-flow basis. Furthermore, the temporal aggregation mitigates the bursty characteristics of the traffic flows. Therefore, a prediction of short high-volume traffic peaks, which are one of the main perpetrators of network congestion, is not possible. During this master thesis, it was investigated if the prediction of single traffic flows can also be achieved when using flows that possess the bursty characteristics of real network traffic. For computing the predictions, the recently developed Generalized Kernel Kalman Filter [39], which embeds the formulations of the traditional Kalman Filter into a Reproducing Kernel Hilbert Space, was used. The results of our prediction experiments show that by transforming the traffic flows into the frequency domain, the peak structures of unseen flows can be predicted after learning their key characteristics from flows that stem from the same socket-to-socket connection.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations

Notation	Description
ANFIS	Adaptive Neuro-Fuzzy Inference System
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
ARMA	Autoregressive Moving Average
ARP	Address Resolution Protocol
BLRNN	Bilinear Recurrent Neural Network
CLI	Command-Line Interface
CRAM	Cognitive Routing Algorithm Module
CRE	Cognitive Routing Engine
DFT	Discrete Fourier Transform
D-ITG	Distributed Internet Traffic Generator
ECMP	Equal-Cost Multi-Path
EKF	Extended Kalman Filter
EM	Expectation-Maximization
FFT	Fast Fourier Transform
FT	Fourier Transform
GARCH	Generalized Autoregressive Conditional Heteroskedasticity
GKKF	Generalized Kernel Kalman Filter
GRE	Generic Routing Encapsulation
HMM	Hidden Markov Model
HTTP	Hypertext Transfer Protocol
IDT	Inter Departure Time
IP	Internet Protocol
JSON	JavaScript Object Notation
KF	Kalman Filter
KKF	Kernel Kalman Filter
KKF-CEO	Kernel Kalman Filter based on the Conditional Embedding Operator
KVM	Kernel-based Virtual Machine
MFT	Multiple Flow Tables
MLP	Multilayer Perceptron
OS	Operating System
OVS	Open vSwitch

OVSDB	Open vSwitch Database Management Protocol
PC	Principal Component
PCA	Principal Component Analysis
QoS	Quality of Service
RBF	Radial Basis Function
RKHS	Reproducing Kernel Hilbert Space
RNN	Random Neural Network
SDN	Software-Defined Network
SNMP	Simple Network Management Protocol
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TE	Traffic Engineering
ToR	Top of Rack
UDP	User Datagram Protocol
UKF	Unscented Kalman Filter
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNC	Virtual Network Computing
WAN	Wide Area Network

1 Introduction

Today's data centers used by private enterprises or the public sector provide a variety of applications and cloud-based services. Since the services differ considerably in their traffic characteristics, it is a difficult endeavor for the underpinning network structures to handle the traffic well during different network situations. One typical problem that arises in networks that have to deal with large amounts of traffic is congestion. When a network device is receiving more data packets than it can process, the packets are delayed or lost which in turn reduces the overall throughput of the network. This inevitably lowers the performance of the services and leads to a dissatisfying end-user experience.

Network protocols like the Transmission Control Protocol (TCP) provide congestion control mechanisms to cope with the problem of traffic congestion. Whenever a packet loss is detected at the sender side of a TCP connection, a congestion in the network is assumed and the packet transmission rate drastically reduced. However, until the loss is noticed from the sender, many packets are already lost which, together with the decreased transmission rate, negatively influences the quality of the provided services and lowers the network throughput.
One way to deal with congestion on the hardware side is to deploy special network topologies like the fat-tree topology that tries to prevent link overutilization by providing bigger bandwidths on links higher in the network hierarchy that need to transmit the aggregated traffic from the lower levels. However, providing an oversupply in bandwidth is expensive and as shown by Benson et al. [6], packet loss in data centers is primarily happening on links that are not heavily utilized on average. Hence, increasing the bandwidth would not solve the problem. The reason for the losses can be found in the bursty nature of the network traffic which causes congestion when multiple traffics flows transmitted on the same link produce high peaks simultaneously.
Instead of restricting the transmission rates of single TCP connections reactively after a loss appeared, another approach would be to spread the existing flows over the network to better exploit the provided bandwidth capacities and to mitigate the influence of traffic bursts. Traffic Engineering (TE) systems that implement load-balancing functionalities try to achieve this by computing optimal routing paths for traffic flows. A commonly used routing strategy is Equal-Cost Multi-Path (ECMP) [7] routing which computes multiple best paths to a destination on a per-hop basis and randomly assigns flows to them. However, since the flow-to-path mapping does not take the flow sizes into account, congestion situations still arise in the presence of high-volume flows, called elephant-flows, because they are potentially send over the same link. Additionally, the routing paths are static and therefore, ECMP cannot react on changing traffic characteristics.
In general, traditional network architectures are not well-suited for developing sophisticated TE systems because they miss a set of desired properties. No entity in the network is able to easily collect statistical information from all network devices and to aggregate them to form a global view of the network which would allow a TE application to understand the current network situation and simplify the computation of routing paths. Moreover, the routing behavior in traditional networks is rather static and cannot be altered programmatically on a short notice. This makes it practically impossible to react to changing traffic characteristics. The relatively new network architecture concept of Software-Defined Networking possesses all the mentioned properties. Networks implementing this concept, from now on called Software-Defined Networks (SDNs), separate the network control from the forwarding functions of the network. The central entity is thereby represented by the SDN controller which is connected over the control plane to the network devices which in turn form the data plane. This setup allows the applications running on the controller to collect information from the network devices and alter the routing behavior of the network in real time. Because of their suitability to conduct TE in them, numerous different TE systems were already developed for preventing congestion in SDNs. Yet, all approaches measure the current flow sizes and use this information for the path computation. Therefore, they can only react to the current situation which is why congestion can not be prevented in the presence of short flow bursts.
An optimal TE system would be capable of predicting the traffic flows so that a overutilization situation would be detected before it could be measured on the network link. Previous research on network traffic prediction mostly focused on traffic data which was aggregated on a temporal scale and by combining many single flows. Since congestion in data centers is caused by the interaction of short traffic bursts from elephant flows, the proposed prediction methods cannot be utilized by potential new TE systems that aim to predict traffic flows in SDNs.

During this master thesis, it was investigated if the course of single elephant flows can be predicted on a small time scale which would allow a TE system to handle the bursty nature of network traffic in the context of congestion control. In a primary step, real-world traffic data from a university data center was analyzed in order to find structures in the diverse flow set that could be learned. Eventually, repeating structures were found for flows stemming from recurring socket-to-socket connections. Subsequently, the selected traffic data was used in traffic prediction experiments, whereas

the Generalized Kernel Kalman Filter (GKKF) concept was chosen as the machine learning approach dealing with the learning and prediction problem. The results from this master thesis can be seen as first steps in solving the task of revealing, if a SDN-based TE system capable of small time scale single-flow traffic prediction is feasible. Furthermore, in the course of the prediction experiment preparation, a network experiment framework was developed which can be used to conduct arbitrary experiments with simulated traffic and custom topologies in a scalable virtualized SDN.

1.1 Thesis Structure

In Section 2 previous research related to this master thesis is presented which includes work about TE in SDNs, the prediction of network traffic and other methods than the GKKF that can be used for state estimation in non-linear systems. Section 3 begins with explaining the concept of Software-Defined Networking. Subsequently, the foundations needed for the GKKF are laid which comprise time series modeling and Reproducing Kernel Hilbert Spaces. Using the presented foundations the formulations of the GKKF are derived in Section 4. In Section 5 the developed experiment framework used to emulate a SDN is presented by showing its components and how a typical workflow of the framework looks like. The results of the traffic prediction experiments using the GKKF are shown in Section 6, whereas prior to that the experimental data and needed preprocessing steps are described. Conclusions about the results of this master thesis are finally drawn in Section 7.

2 Related Work

In the following chapter, some selected already conducted research related to this master thesis is summarized. Starting with research on TE systems specifically designed for SDNs, followed by work on the general case of network traffic prediction. In the last section, research related to the Generalized Kernel Kalman Filter (GKKF), which was used during this master thesis for estimating the state of a non-linear system, is presented.

2.1 Traffic Engineering in Software-Defined Networks

The emerging network architecture concept of Software-Defined Networking possesses unique characteristics that make SDNs well-suited for developing sophisticated TE systems. Therefore, a solid amount of research was already conducted in this area. A TE framework usually consists of a traffic measurement and a traffic management component [8]. The traffic measurement component collects status information from the network that could include information about the network structure, its current performance or about the traffic transmitted through it. The traffic management component, which runs at the SDN controller, utilizes this information to equip the network with certain desired functionalities by implementing them as network services. The existing TE systems presented in this chapter focus on maximizing the throughput of the network by providing a general load-balancing functionality in data centers or similar networks. Moreover, Software-Defined Networking architectures were also used to achieve higher link utilization in Wide Area Networks (WANs) in which multiple data centers are interconnected [9, 10].

Agarwal et al. [11] showed that the problem of computing the optimal paths in a SDN that maximizes the network throughput can be formulated as a multi-commodity flow problem. Assuming that traffic flows can be split over multiple paths, the resulting optimization problem can be solved in polynomial time by the controller using fully polynomial time approximation schemes. After the optimal paths are found, the controller forces the network devices to route the traffic flows accordingly.
However, in real network scenarios computing and installing paths for every single flow can easily exceed the computational capacities of the controller. Therefore, other approaches like the dynamic flow scheduling system Hedera [12] use flow information collected at the network devices to identify large flows in the network, called elephant flows, which can cause link over-utilization when sent over the same path. The central controller is calculating non-conflicting paths only for those high-volume traffic flows and instructs the network devices to re-route them accordingly. Another TE system called Mahout [13] is also detecting elephant flows. However, in contrast to Hedera the identification of an elephant flow is not done by polling network information from the network devices but rather at the end host sides by monitoring the hosts' socket buffers. Thus, the overhead of control traffic sent to the central controller is reduced. Moreover, elephant flows can be detected earlier and thus, rerouting can be achieved faster. The scalable flow management tool DevoFlow [14] was designed with the same goals in mind. Here, control for smaller traffic flows is devolved back to the network devices. By using matching rules based on wildcards routing is directly handled by the network devices by default. Flow statistics, which are again collected locally, are used to detect elephant flows whereas the controller is only utilized to make re-routing decisions for those flows.
Another TE system that tries to find near-optimal routing paths, while keeping the overhead of control traffic low, is the Cognitive Routing Engine (CRE) [15]. Its Cognitive Routing Algorithm Module (CRAM) computes the most suitable paths for new flows in the network using Random Neural Networks (RNNs) together with reinforcement learning. One RNN is constructed for every network device present in the shortest path initially calculated for a new flow. CRAM makes its decisions upon network link characteristics. Therefore, in contrast to the TE systems described before, network information only has to be collected for every network link instead of every flow. Furthermore, the network monitoring process is optimized by storing the measured characteristics in a database for reusing and sharing them between the different RNNs. Thus, CRE provides a TE system with a small monitoring overhead. However, the system is not focusing on large traffic flows and therefore path computation and installation again needs to be handled by the controller for every new flow in the network.

All TE systems described share a unifying characteristic. They solely rely on network information that, in the optimal case, describe the current network situation and therefore, the systems can only *react* to those situations. As shown in [6], traffic in data centers is bursty in nature. Thus, flows switching from an OFF to an ON phase can suddenly produce high traffic loads. If several high-volume flows peak at the same point in time this can lead to congestion in the network. In such a scenario it is not enough to start the path computation and installation process when the high traffic loads are

measured, because during the execution of the process data packets are already lost. Instead, it would be useful to be capable of predicting traffic flows to become aware of upcoming over-utilization situations. One TE system developed for data centers that tries to separate the aggregated flows between Top of Rack (ToR) switch pairs or sever pairs in short-term predictable and non-predictable flows is MicroTE [16]. For the predictable flows the controller is used to compute optimal paths, whereas the unpredictable flows can only be routed using ECMP as a fallback routing strategy. The term *predictable* is used by the authors for flows whose volume does not differ from the mean traffic of the associated ToR switch pair by more then 20%. Therefore, the progression of a traffic flow is not predicted, it is just assumed that its load will stay fairly constant in the next 1-2 seconds. The authors show that in some data centers a substantial amount of traffic remains constant on a short time scale. However, they also mention that the percentage of predictable traffic heavily depends on the applications that run in the data center. Thus, on average of all surveyed data centers only 35% of the traffic between ToR switch pairs was predictable. MicroTE is only achieving good performance when routing traffic at the finer spatial granularity of server pairs. For this scenario the authors assumed that the traffic between a server pair is predictable if this accounts for the associated ToR switch pair. However, this might not be the case because some of the traffic consistency could be caused solely by the aggregation process. Therefore, it can be concluded that there is a need for an actual prediction of the traffic flow progression. A TE system could use these predictions to produce a forecast on the network situation and initiate re-routing early enough to prevent link over-utilization, even in networks with dynamic traffic patterns.

2.2 Network Traffic Prediction

As described in the last section, current TE frameworks developed for SDNs only react upon the network situation instead of predicting its future state. However, the general case of network traffic prediction was already subject of extensive research. One of most simple approaches usable to model a time series of traffic data is the Autoregressive Moving Average (ARMA) model which consists of an autoregressive part performing a regression on the series of data points and a moving average part that tries to model the error of the time series. In the literature the ARMA model was used to predict network traffic mostly from single applications like bit torrent [17] or file transfer applications [18]. Since the ARMA model is only applicable for time series produced by stationary stochastic processes, the Autoregressive Integrated Moving Average (ARIMA) model was developed. It provides a generalization of the ARMA model since it can also work with data streams from non-stationary processes by performing a differencing step on them. The ARIMA model and variants of it were used in different scenarios for traffic prediction, e.g. in 3G mobile networks [19] or public safety networks [20].

ARMA and ARIMA models are only applicable for linear time series with constant variance. To model non-linear time series with a time dependent variance the Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model can be used. In [21] the authors showed that the model performs better than the ARIMA model in capturing the bursty nature of Internet traffic which produces a variance change over time. The group of neural networks also allows to model non-linear time series, which is why already many different neural network types were tested for predicting network traffic. In [22] e.g. dynamic Bilinear Recurrent Neural Networks (BLRNNs) were applied and showed superior performance compared to static BLRNNs or classical neural networks like the Multilayer Perceptron (MLP), which were used in [23] and [24] for the prediction task. Furthermore, a special MLP variant called neural trees that allows for over-layer connections and different activation functions for different network nodes was used in [25]. Instead of applying other neural network types, some research focused on combining neural networks with linear approaches like the ARIMA model because the authors assume that the traffic time series consists of linear and non-linear components [26]. Other researchers also connected neural networks with other system methodologies like fuzzy systems, thereby developing the Adaptive Neuro-Fuzzy Inference System (ANFIS) [27]. Moreover, also Support Vector Machines (SVMs) were used for predicting network traffic, e.g. by Liang et al. , whereas their parameters were selected using the ant colony optimization algorithm [28].

The presented non-exhaustive list of related research shows that already numerous different methods were used to predict network traffic. However, none of the described approaches could be directly integrated into a TE system that aims at conducting live predicting and rerouting of traffic flows in networks like data centers on a fine-grained time and flow scale. The reason is that the used data was always aggregated in two ways.
Firstly, the applied data sets mostly consisted of traffic data which was collected during a time span of days, weeks or even months and which was aggregated on a temporal scale leading to sampling intervals for the traffic load between 1 second and 1 hour. Only in [25] an interval of 0.1 seconds was used. By aggregating the traffic data in this manner its bursty appearance is mitigated. The time series become less complex and much easier to learn. However, since fine-grained characteristics are not present in the data set anymore, they cannot be predicted, at least from the approaches performing a high temporal aggregation. This is a problem because already rather short traffic peaks with very high loads can cause congestion in a network, which was also observed by Benson et al. [6].

Secondly, the selected data was also aggregated in terms of flows, meaning that usually all independent TCP and User Datagram Protocol (UDP) flows representing a single socket-to-socket connection were combined to one big flow. Again, this hides the bursty characteristics of the traffic and prevents the prediction of single flows. As a result, a TE system could not identify single high-volume flows in the network and implement a fine-grained rerouting functionality which, as shown in [16], can have a decisive influence on the performance of the TE system. Predicting single flows again creates the need for small sampling intervals because otherwise, short high-volume flows would be represented by only a few data points.
In all of the mentioned approaches flows were split into a training and test part, therefore predictions were made based on the history of the flows. As a result, the model for the prediction would have to be learned during the flow life time which is again very difficult for short high-volume flows.
In contrast to the already conducted research, a different approach was examined during this master thesis. Firstly, the available traffic data was used at a more fine-grained level in terms of flow and sampling interval aggregation. Secondly, flows were not predicted based on their own history. Instead, entire flows with a similar structure that appeared earlier in the network served as the training data. Since the complex flow structures are produced by a process about which we have no knowledge and whose state is not directly observable for us, the powerful concept of Hidden Markov Models (HMMs), which is described in more detail in Section 3.2.1, was utilized to model the underlying process.

2.3 State Estimation of Non-linear Systems

For predicting the upcoming load of a traffic flow in a SDN the underlying stochastic process producing the flow and forming its appearance has to be modeled and its hidden state estimated. One well-known algorithm for state estimation that uses a HMM to model the system is the Kalman Filter (KF) [29]. However, since the KF can only be used for linear systems, extensions like the Extended Kalman Filter (EKF) [30] and the Unscented Kalman Filter (UKF) [31] were developed to allow state estimation also for non-linear systems. In the original KF the model is represented by linear functions that are assumed to describe the dynamics of the system. For the EKF and UKF also non-linear functions can be used for modeling the underlying process as long as they are differentiable. When computing a new state estimate with the EKF a Taylor series approximation is conducted, thereby linearizing the model around the current working point.
The UKF achieves more stable and robust results using another approach. Instead of approximating the non-linear functions, sampling points are taken from the current state estimation given by a mean and covariance. The sampling points then represent the approximated probability distribution of the state which is transformed by applying the non-linear functions to every point. Both the EKF and UKF assume that the non-linear functions describing the system dynamics are known. In the setup observed during this master thesis the dynamics are unknown and therefore, other methods have to be used.

When the model of the system is not known it has to be learned from data. However, if the true hidden states are not present in the available data set heuristic approaches like the Expectation-Maximization (EM) algorithm are used to learn the model. Often, these algorithms need to be carefully initialized and suffer from local optima issues. A recent approach that allows for learning the model of a HMM without the need for hidden state information is the spectral learning algorithm for HMMs [32]. By using the observable operator model [33], the learned HMM is modeled entirely in observation space. As a result, the transition and observation model of the system are not explicitly recovered but an internal representation of the hidden states is formed which is linearly related to the true representation. Under certain separation assumptions which imply that the different observation probability distributions obtained for every discrete hidden state of the system are distinct [34], the learned HMM in observation space can be used to predict future observations given only a history of observations.
Since the spectral learning algorithm can only be used for systems with discrete observations and hidden states, a kernelized version was developed by Song et al. [35] by embedding its formulation in a Reproducing Kernel Hilbert Space (RKHS). The resulting algorithm allows to utilize the advantages of spectral algorithms also for continuous state spaces. However, the assumption of distinct probability distributions is still made. The GKKF computes an uncertainty measure for the prediction of the hidden state which is possible because the embedding of the state estimation is mapped back to the original space. The kernel spectral algorithm cannot provide a similar measure because the state estimation is represented by a sample in the training data that maximizes the *a posteriori* belief. In [36] experiments were conducted showing that the GKKF outperforms the kernel spectral algorithm in a simulated and a real-world scenario.

Other approaches for non-linear time series modeling that are closer related to the GKKF are the Kernel Kalman Filter (KKF) [37] and the Kernel Kalman Filter based on the Conditional Embedding Operator (KKF-CEO) [38]. The KKF is a kernelized version of the KF where the observations, system states and update equations of the KF are brought to a feature space by using a kernel function. However, in contrast to the GKKF a kernelized version of the system model is only derived for the transition model but not for the observation model. As a result, the observations are computed

from the states simply by adding a noise term. Another difference to the GKKF is that the KF formulations are only embedded in a sub-space of the feature space and hence, the approach is not fully exploiting the infinite-dimensionality characteristic of the feature space.
The KKF-CEO embeds the formulations of the KF in a RKHS by using the conditional embedding operator which is explained in more detail along with RKHSs in Section 3.3. In contrast to the KKF the KKF-CEO formulates the KF in the full feature space provided by the kernel. Moreover, the transition model does not have to be learned using the EM algorithm, as it is done with the KKF, but can be computed from the training data. However, as for the KKF the observation model is not formulated in the RKHS and the observations again interpreted as noisy variants of the system states. Computing the transition model under this assumption using the embeddings of the noisy observations is not fully valid from a theoretical point of view since the observations were not generated by a Markov process. Furthermore, this leads to update equations that are farer away from the original KF equations compared to the GKKF. Besides, in [39] it was shown that the GKKF outperforms the KKF-CEO in different experiments using data from simulated environments and real-world applications.

3 Foundations

In this chapter, the concept of Software-Defined Networking and its advantages compared to traditional network architectures are described. Software-Defined Networking serves as the foundation for building the experiment framework introduced in Chapter 5. Furthermore, the fundamentals needed for the GKKF from Chapter 4 are explained.

3.1 Software-Defined Networking

Software-Defined Networking is an architecture concept for networks that should enable them to meet the requirements of modern computing environments like data centers or carrier networks. Traditional static network architectures are ill-suited in this manner because they cannot cope with the dynamically changing need for memory and computing resources of today's network services. The reasons for this change in requirements can be found, among others, in the rise of cloud services, the special needs of big data applications and the change in network traffic patterns from solely client-server based to more machine-to-machine communication. In Section 3.1.1 the main components of a Software-Defined Networking architecture are described, followed by a more detailed view in Section 3.1.2 on how the routing behavior in a SDN can be altered.

3.1.1 Software-Defined Networking Architecture

The core idea behind Software-Defined Networking is the separation of the network control from the forwarding functions of the network. A high-level view of the architecture is shown in Figure 3.1. The infrastructure layer, which is also called data or forwarding layer consists of network devices that are connected following a certain topology. In traditional networks the network control functionality is implemented in the network devices, making use of vendor specific operating systems, applications and protocols.
In a SDN the infrastructure is abstracted by moving network control to a central entity in the network, the SDN controller. The controller forms the control layer of the network and communicates over a vendor-neutral protocol with the network devices. The first protocol alike, called OpenFlow, is developed by the Open Network Foundation [40]. The controller can collect statistical information from the network devices, use them in its decision making process and alter the routing behavior of every network device, which will be further described in Section 3.1.2. As a result of the abstraction, network devices can remain simple and offer open interfaces to the higher layers which differs from today's highly-specialized vendor dependent devices. Simpler network devices will reduce the overall complexity of the network and make it easier to add new devices to the network or relocate existing ones. Subsequently, networks will become more flexible and highly scalable.
The top layer is the application layer which abstracts the controller functionalities to make them callable for business applications. Thus, network control becomes directly programmable in a more convenient and vendor independent way which accelerates the development of new network services. Moreover, the centrality of the controller makes it possible to gain a global view over the network and to use this knowledge to build applications that can alter the network behavior in real time.

3.1.2 OpenFlow

As mentioned before, the communication between network devices and the controller is handled over the open standard protocol OpenFlow which is implemented at both endpoints and currently available in version 1.5.1 [41]. Its main features are the extraction of statistical information from the network devices and its ability to directly manipulate their routing behavior.
In OpenFlow all data packets transmitted over the network that match a certain set of criteria are per definition one unique flow. As matching criteria packet properties like its destination Internet Protocol (IP) address, its source IP address, its TCP destination port and many more can be used. The filtering for these criteria is happening at the network device where every unique flow is represented by an entry in a so-called flow table, which form a key element of an OpenFlow enabled network device. Flow tables, as illustrated in Figure 3.2, can be seen as a replacement for routing tables in traditional devices. When a new packet is received from the device, the flow table is filtered using the match fields until a fitting entry is found. Then the specified action is executed. Using an example from Figure 3.2, the device would forward all packets that have the destination IP address 5.6.7.8 out of port 2. Instead of forwarding a packet to another

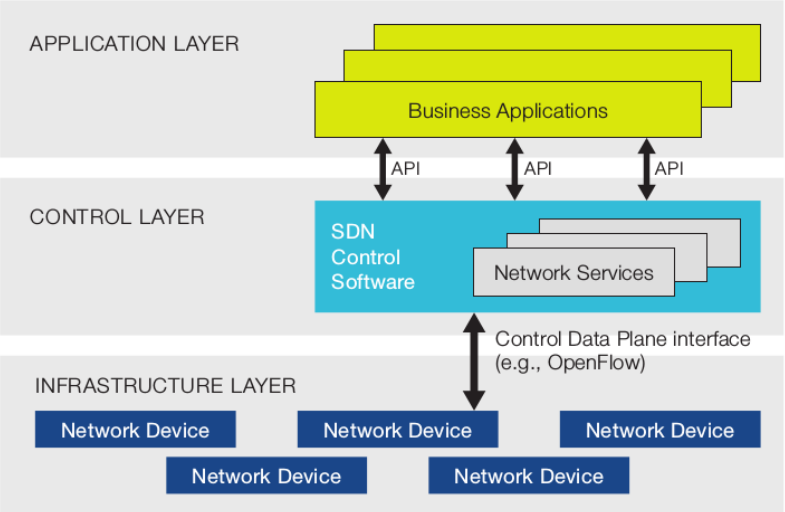
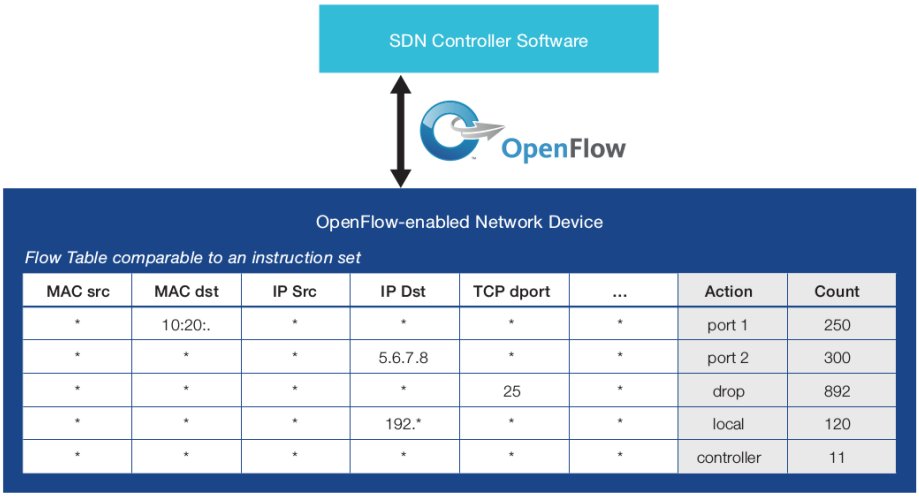


Figure 3.1.: Software-Defined Networking Architecture [1]

network device it can drop it, alter the packet values or forward it internally. The complete list of all possible actions can be found in [41]. It is also possible to set a priority number for each entry that decides which flow entry gets picked if multiple ones match the criteria. Therefore, a flow table allows to program the behavior of the forwarding plane. This complete control over the forwarding functionality of the abstracted infrastructure devices makes SDNs flexible instead of monolithic.
Using one flow table per network device is simple but also very restrictive and leads to a high number of flow entries if a more fine grained flow separation is required. This is why with OpenFlow 1.3 support for Multiple Flow Tables (MFT) was introduced. MFT allows it to build a multi-step processing model where the first flow table is used to preprocess data packets by filtering for one property and subsequently forwarding the packet to another flow table where further matching and processing occurs. Thus, MFT allows to handle more interesting real-world network situations like port-based Virtual Local Area Network (VLAN) identifiers or Virtual Routing and Forwarding [42].

Another element of OpenFlow that allows for additional packet forwarding methods are group tables to which packets can be forwarded to from flow tables. Like flow tables they consist of single entries but instead of assigning only one action to one entry a set of so-called action buckets is defined. Every bucket can consist of several individual actions and associated parameters. The type of a group entry decides whether all action buckets belonging to that entry are executed or just some of them. However, always all actions of one action bucket are executed. Group tables are very useful when several actions should be performed on flows like a modification of the packets followed by a forwarding action. Moreover, group tables can also be used to split flows and forward certain shares of the flow to different ports. Thus, flows can be split over different routing paths in the network.
The third type of tables in an OpenFlow enabled network device are meter tables which can be used to limit the bandwidth of single flows and thus, offer a simple already build-in Quality of Service (QoS) functionality. The bandwidth rates are defined in so-called meter bands together with the information on how a packet should be processed if the associated flow exceeds that rate. Multiple meter bands can now be assigned to a single entry in the meter table and those meter entries can in turn be assigned to flow entries. The implementation of more complex QoS applications can be achieved by the combination with another OpenFlow property, namely QoS queues. Here every port of a device can be assigned to one or multiple queues that have a certain minimum and maximum bandwidth rate configuration. Therefore, in contrast to the meter tables this QoS mechanism works on a per-port basis instead of a per-flow basis.

As mentioned before, another key feature of OpenFlow is the extraction of statistical information from the network devices which can be used by the controller applications in their decision making process. The statistical information is presented as counters that exist for every flow table, flow entry, port, queue, group, group bucket, meter and meter band. Coming back to the example shown in Figure 3.2, whenever a packet is received from the device with the IP destination address 5.6.7.8 the associated counter will be increased by one. These counters do not only exist for received packets, but also for transmitted packets, dropped packet, packet errors and many more. The complete list of all available counters in OpenFlow can be found in Appendix A.1.



MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Figure 3.2.: OpenFlow Flow Table [1]

3.2 Time Series Modeling

The GKKF described in Chapter 4 has to deal with a time series of statistical information. For being able to make predictions upon those series of data points the processes of the underlying system have to be modeled. The HMM which is utilized by the GKKF for that purpose is introduced in Section 3.2.1, followed by the KF in Section 3.2.2 which forms the foundation of the state estimation technique used in the GKKF.

3.2.1 Hidden Markov Model

The superset of Markov Models can be used to model stochastic processes were it is assumed that the process holds the Markov property. This means that the state of the associated system, which needs to be discrete, depends only on a certain number of preceding states. Accordingly, if the current state depends only on the last state, the process is said to possess a first-order Markov property. Depending on the setup of the system environment, different types of Markov Models can be formulated. If the system cannot be controlled and therefore acts autonomously and if the state of the system is only partially observable, the used Markov Model is called Hidden Markov Model (HMM) because the true state of the system is hidden. HMMs have been utilized in a wide range of different research fields over the last decades to solve problems, e.g. in computational biology [43] or for pattern recognition in speeches or gestures [44, 45].

A HMM can be defined as a 5-tuple $\Omega = (X, Y, \boldsymbol{\delta}, \boldsymbol{\pi}, \lambda)$. The state space of the system is denoted by X, whereas $x_t \in X$ represents the state at time point t. As described before, the true state of the system cannot be observed, only an observation emitted by the system is available and denoted by $y_t \in Y$. Observations can take a discrete or continuous value. When assuming a first-order Markov property the state probability of x_t is given by $p(x_t|x_{t-1})$. The transition probabilities between the states are described by the transition matrix $\boldsymbol{\delta}$ and for the initial state by $\boldsymbol{\pi}$. The emission or observation probabilities $p(y_t|x_t)$ are described by the matrix $\boldsymbol{\lambda}$. Both matrices together form the parameters of the model $\theta = \{\boldsymbol{\delta}, \boldsymbol{\lambda}\}$.
One of the problems usually formulated when a stochastic process is modeled by a HMM is filtering. The goal is to calculate a probability distribution estimation of the system state $p(x_t|y_{1:t}; \theta)$, given only the parameters of the model θ and a sequence of observations up to the current time point denoted by $y_{1:t}$. The most probable state can be found by calculating the joint probability of the current state x_t and the observation sequence $y_{1:t}$, which is given by the sum over the joint probability of x_t, $y_{1:t}$ and all possible preceding states x_{t-1} resulting in

$$p(x_t, y_{1:t}) = \sum_{x_{t-1}}^{X} p(x_t, x_{t-1}, y_{1:t}). \tag{3.1}$$

After expanding the term inside the sum by using the chain rule it can be simplified. This is possible because the assumed Markov property of the HMM implicates that x_t solely depends on x_{t-1} and y_t therefore only on x_t. The resulting recur-

sive formulation

$$p(x_t, y_{1:t}) = \sum_{x_{t-1}}^{X} p(y_t|x_t, x_{t-1}, y_{1:t-1}) p(x_t|x_{t-1}, y_{1:t-1}) p(x_{t-1}, y_{1:t-1}) \tag{3.2}$$

$$= p(y_t|x_t) \sum_{x_{t-1}}^{X} p(x_t|x_{t-1}) p(x_{t-1}|y_{1:t-1}) \tag{3.3}$$

can e.g. be evaluated efficiently by making use of the forward algorithm [46]. Besides filtering another often formulated problem is the probability estimation of a whole state sequence $p(x_{1:T}|y_{1:T}; \theta)$ given an observation sequence and the model parameters. This task is called decoding and the most probable sequence can again be found by calculating the joint probability with [47]

$$p(x_{1:T}, y_{1:T}) = p(y_{1:T}|x_{1:T}) p(x_{1:T}) \tag{3.4}$$

$$= p(y_1|x_1) p(x_1) \prod_{t=2}^{T} p(y_t|x_t) p(x_t|x_{t-1}), \tag{3.5}$$

which can be accomplished with the Viterbi algorithm [46].
In both formulated problems, the model parameters θ were given. However, often they are not known *a priori* and have to be learned from a set of training samples. A classic approach to solve this problem is the EM algorithm derived for HMMs, also known as the Baum-Welch algorithm [48]. In recent years spectral learning algorithms were developed that do not suffer from issues with local optima [32, 35].

3.2.2 Kalman Filter

In the case of a continuous state space the methods described in the last section cannot be utilized. One technique that can be used instead for state estimation problems in Bayesian models with a continuous state space is the Kalman Filter (KF) [29]. The KF can be used for linear systems and models them with a state-transition model represented by the system matrix $\mathbf{F}$ and an observation model represented by the observation matrix $\mathbf{H}$. The system dynamics are then given by

$$x_{t+1} = \mathbf{F} x_t + v, \qquad y_t = \mathbf{H} x_t + w, \tag{3.6}$$

whereas x_t represents the state of the system, y_t the observation emitted from the system, v the process noise and w the observation noise. Both noise terms are assumed to be drawn from a multivariate Gaussian distribution with zero mean and the covariances P and R respectively. The system matrix $\mathbf{F}$ is formed out of equations that are assumed to describe the dynamics of the system and the observation matrix $\mathbf{H}$ out of equations that describe how an emitted observation depends on the state of the system. After defining the models the state can be estimated, whereas the state is represented by a mean μ_x and a covariance Σ_x which measures the uncertainty of the current estimate.
The KF algorithm consists of two update procedures that are executed iteratively. In the **prediction update** the system state is estimated using only the state estimate of the last time step and the state-transition model, whereas a superscript dash in the notation denotes a state estimation that was done before (*a priori*) receiving an observation at the given time point t.

$$\mu_{x_t}^{-} = \mathbf{F} \mu_{x_{t-1}}, \qquad \Sigma_{x_t}^{-} = \mathbf{F} \Sigma_{x_{t-1}} \mathbf{F}^T + P. \tag{3.7}$$

After an observation is emitted by the system, the **innovation update** step of the KF algorithm is executed. The observation is used to update the *a priori* mean by adding an error term from the observation model multiplied with the Kalman gain matrix $\mathbf{Q_t}$. The covariance prediction is updated using only $\mathbf{Q_t}$ and the observation matrix $\mathbf{H}$. Therefore, it can already be calculated before the observation is received.

$$\mu_{x_t} = \mu_{x_t}^{-} + \mathbf{Q_t} (y_t - \mathbf{H} \mu_{x_t}^{-}) \qquad \Sigma_{x_t} = \Sigma_{x_t}^{-} - \mathbf{Q_t} \mathbf{H} \Sigma_{x_t}^{-} \tag{3.8}$$

The same holds for the calculation of the Kalman gain matrix. The Kalman gains can be seen as weights that decide how much the updated mean μ_{x_t} follows the predicted mean $\mu_{x_t}^-$ or the observation y_t and are calculated with

$$\mathbf{Q_t} = \Sigma_{x_t}^- \mathbf{H}^T (\mathbf{H} \Sigma_{x_t}^- \mathbf{H}^T + R)^{-1}. \tag{3.9}$$

The prediction and innovation update steps are usually alternated. However, it is also possible that the observations arrive irregularly and so the state has to be predicted for several time steps in a row.
As mentioned before, the KF assumes linear system dynamics which limits its usage substantially. For non-linear systems several other KF variants were developed, like the EKF [30] or the UKF [31]. However, as described in Chapter 2 both methods require known system dynamics and do not work well for systems with high-dimensional observations. Another more promising group of KF variants is the one of Kernel Kalman Filters to which also the GKKF presented in Chapter 4 belongs.

3.3 Reproducing Kernel Hilbert Space

Since the original KF can only deal with linear systems, the GKKF uses a kernelized KF variant which allows to conduct state estimation for systems with unknown system dynamics. This section introduces kernel methods in Section 3.3.1 and reproducing kernels in Section 3.3.2 associated with RKHS. In Section 3.3.3 it is then explained how distributions can be embedded in a RKHS. This forms an important foundation for the kernelized KF introduced in Chapter 4.

3.3.1 Kernel Methods

Kernel methods are a group of algorithms used in the field of machine learning to find structures or patterns in data sets and utilize this knowledge for solving a wide range of different tasks like classification or regression analysis. Two state-of-the art kernel methods are support vector machines [49] and Gaussian processes [50].
The unifying property that all kernel methods share is that the used data is implicitly brought from its original space to a possibly infinite-dimensional feature space. Non-linear relationships between the data points can then be represented as linear ones in the higher-order space and thus, linear algorithms can be used to learn the existing structure in the data. Many different functions could be used for the mapping into feature space, e.g the Radial Basis Function (RBF) $\phi(x) = \exp\left(-\frac{||x-c||^2}{2b^2}\right)$. In the given equation x represents a data point in the original space and $\phi(x)$ its equivalent in the feature space. The characteristics of the RBF are defined by its bandwidth b and its center c.
Calculating the feature mapping explicitly, e.g. when evaluating the inner product of two feature vectors $\varphi(x_1)^T \varphi(x_2)$, whereas $\varphi(x_1)^T = [\phi_1(x_1), \phi_2(x_1), \ldots, \phi_d(x_1)]$, would be computationally very expensive and only possible for a feature space with a finite dimension d. Instead, a kernel function can be defined which calculates the inner product without the need of actually computing the mapping. Again, a scalar is returned which can be seen as a similarity measure between the two input data points. One often used kernel is the Gaussian kernel which is defined as,

$$\begin{aligned} k(x_1, x_2) &= \langle \varphi(x_1), \varphi(x_2) \rangle && (3.10)\\ &= \varphi(x_1)^T \varphi(x_2) && (3.11)\\ &= \exp\left(-\frac{||x_1 - x_2||^2}{2b^2}\right). && (3.12) \end{aligned}$$

This approach is called the *kernel trick* and allows to use a feature space with an arbitrary number of dimensions. When the similarity measures should be calculated for a whole data set $X = \{x_1, x_2, \ldots, x_m\}$ the kernel matrix $\mathbf{K}_{xx} \in \mathbb{R}^{mxm}$ is formed,

$$\mathbf{K}_{xx} = \mathbf{\Upsilon}_x^T \mathbf{\Upsilon}_x = \begin{bmatrix} \varphi(x_1)^T \\ \vdots \\ \varphi(x_m)^T \end{bmatrix} \times [\varphi(x_1), \ldots, \varphi(x_m)] = \begin{bmatrix} \langle \varphi(x_1), \varphi(x_1) \rangle & \cdots & \langle \varphi(x_1), \varphi(x_m) \rangle \\ \vdots & \ddots & \vdots \\ \langle \varphi(x_m), \varphi(x_1) \rangle & \cdots & \langle \varphi(x_m), \varphi(x_m) \rangle \end{bmatrix}. \tag{3.13}$$

In theory all valid kernel functions $k(x_i, x_j) = \langle \varphi(x_i), \varphi(x_j) \rangle$ are symmetric and all kernel matrices $\mathbf{K}$ with the elements $\mathbf{K}_{ij} = k(x_i, x_j)$ are positive semi-definite. Therefore, the kernel function satisfies Mercer's condition which is

$$\iint_{ij} f(x_i)\, k(x_i, x_j)\, f(x_j)\, \mathrm{d}i\mathrm{d}j \geq 0, \tag{3.14}$$

for any squared integrable function $f(x)$ [51]. The kernel matrix $\mathbf{K}$, also called Gram matrix, then contains the inner products of all data point pairs.

3.3.2 Reproducing Kernels

Given a kernel function k on a data set $X = \{x_1, x_2, \ldots, x_m\}$ defined as,

$$\begin{aligned} k(x_i, x_j) &= \langle \varphi(x_i), \varphi(x_j) \rangle && (3.15)\\ &= \langle k(x_i, \cdot), k(x_j, \cdot) \rangle && (3.16)\\ &= \langle k_{x_i}, k_{x_j} \rangle && (3.17)\\ &= \mathbf{K}_{ij}. && (3.18) \end{aligned}$$

When calculating the Gram matrix $\mathbf{K}$ the resulting inner products form an inner product space H with an arbitrary number of dimensions. If the space is complete and thus contains the inner products of all data point pairs it is called a Hilbert space. The elements of the Hilbert space H can also be seen as functions $f(\cdot) = k(x, \cdot) = k_x$ that produce the inner product and are approximated with the equation

$$f(x) = \sum_{i=1}^{\infty} a_i\, k(x_i, x), \tag{3.19}$$

which represents a weighted sum of kernel evaluations with the weights $a_i \in \mathbb{R}$. It can now be shown with

$$\begin{aligned} f(x) &= \sum_{i=1}^{\infty} a_i\, k(x_i, x) && (3.20)\\ &= \left[\sum_{i=1}^{\infty} a_i\, k_{x_i} \right]^T k_x && (3.21)\\ &= \left\langle \sum_{i=1}^{\infty} a_i\, k_{x_i}, k_x \right\rangle && (3.22)\\ &= \langle f, k_x \rangle && (3.23) \end{aligned}$$

that the kernel has the so-called reproducing property. By building the inner product of function f and the kernel evaluated at x, f gets reproduced, evaluated at x.
As stated by the Moore–Aronszajn theorem [52], every symmetric positive definite kernel possesses the reproducing property and is therefore called a reproducing kernel. The associated Hilbert space is then called a Reproducing Kernel Hilbert Space (RKHS).

3.3.3 Embed distributions in a RKHS

One foundation for embedding the formulations of the KF in a RKHS is the capability of embedding probability densities. In the work of Smola, Gretton, Song and Schölkopf [53] an approach is described to calculate these embeddings. For a probability density $p(X)$ over a random variable X the RKHS embedding is given as the expected feature mapping

of its random variates as $\mu_X := \mathbb{E}_X[\varphi(X)]$. Whereas $\varphi(X)$ denotes the feature mapping of the random variable with a reproducing kernel function as described in Sections 3.3.1 and 3.3.2. Usually, the underlying distribution is not known but a set of samples from it. The embedding of the distribution, also called mean embedding, can then be estimated through a finite-sample average calculated as

$$\hat{\mu}_X = \frac{1}{m}\sum_{i=1}^{m}\varphi(x_i) = \boldsymbol{\Upsilon}_x^T 1/m. \tag{3.24}$$

Furthermore, the embedding of a joint distribution $p(X, Y)$ is defined as the outer product of the feature mappings of the random variates X and Y as $C_{XY} := \mathbb{E}_{XY}[\varphi(X) \otimes \varphi(Y)]$. The embedding can again be estimated using a finite number of samples from both distributions with

$$\hat{C}_{XY} = \frac{1}{m}\sum_{i=1}^{m}\varphi(x_i) \otimes \varphi(y_i). \tag{3.25}$$

Another form of distribution embedding needed for the GKKF is the embedding of a condition density defined as $\mu_{Y|x} := \mathbb{E}_{Y|x}[\varphi(Y)]$ and introduced in [54]. The embedding of the conditional distribution, given a specific value of x, can be calculated using a conditional embedding operator $C_{Y|X}$ with $\mu_{Y|x} := C_{Y|X}\varphi(X)$, whereas $C_{Y|X} := C_{YX}C_{XX}^{-1}$. Given m samples from both distributions as tuples $m_i = (x_i, y_i)$ the conditional embedding operator can be estimated as

$$\hat{C}_{Y|X} = \boldsymbol{\Upsilon}_{\mathbf{y}}\left(\mathbf{K}_{xx} + \lambda \mathbf{I}_m\right)^{-1}\boldsymbol{\Upsilon}_x^T, \tag{3.26}$$

whereas $\mathbf{K}_{xx} = \boldsymbol{\Upsilon}_x^T\boldsymbol{\Upsilon}_x$ denotes the Gram matrix and $\boldsymbol{\Upsilon}_x = [\varphi(x_1), \ldots, \varphi(x_m)]$ the feature mappings of all states, as already defined in Equation 3.13. The identity matrix of size m given by $\mathbf{I}_m$ is multiplied with the parameter λ to regularize the Gram matrix. With the help of the conditional embedding operator, rules of probability inference like the sum rule, chain rule and Bayes' rule can also be kernelized and used to manipulate the distributions embedded in a RKHS. The marginal distribution of a random variable is given as $p(x) = \int_Y p(x|y)\,p(y)\,\mathrm{d}y = \mathbb{E}_Y[p(x|y)]$. When embedding the distribution $p(x)$ by using the embedding rule for conditional distributions we end up at

$$\begin{aligned}
\mu_X &= \mathbb{E}_X[\varphi(X)] && (3.27)\\
&= \mathbb{E}_Y\,\mathbb{E}_{X|Y}[\varphi(X)] && (3.28)\\
&= \mathbb{E}_Y[C_{X|Y}\varphi(Y)] && (3.29)\\
&= C_{X|Y}\,\mathbb{E}_Y[\varphi(Y)] && (3.30)\\
&= C_{X|Y}\,\mu_Y, && (3.31)
\end{aligned}$$

which gives us the kernelized sum rule. Thus, the conditional embedding operator $C_{X|Y}$ maps the embedded distribution of variable Y to the one of X. We can also use a tensor product feature to embed $p(x)$ resulting in $C_{XX} = \mathbb{E}_X[\varphi(X)\otimes\varphi(X)]$ and the following kernelized sum rule

$$\begin{aligned}
C_{XX} &= \mathbb{E}_X[\varphi(X) \otimes \varphi(X)] && (3.32)\\
&= \mathbb{E}_Y\,\mathbb{E}_{X|Y}[\varphi(X) \otimes \varphi(X)] && (3.33)\\
&= \mathbb{E}_Y[C_{(XX)|Y}\varphi(Y)] && (3.34)\\
&= C_{(XX)|Y}\,\mu_Y. && (3.35)
\end{aligned}$$

With the chain rule $p(x, y) = p(x|y)\,p(y)$ a joint distribution can be formulated as a product of conditional and marginal distribution. The associated embedding $C_{XY} := \mathbb{E}_{XY}[\varphi(X) \otimes \varphi(Y)]$ can then be factorized as

$$\begin{aligned}
C_{XY} &= \mathbb{E}_Y[\mathbb{E}_{X|Y}[\varphi(X)] \otimes \varphi(Y)] && (3.36)\\
&= C_{X|Y}\,\mathbb{E}_Y[\varphi(Y) \otimes \varphi(Y)] && (3.37)\\
&= C_{X|Y}\,C_{YY}, && (3.38)
\end{aligned}$$

whereas C_{XY} is called the cross-covariance operator and C_{YY} the auto-covariance operator. By combining sum and chain rule the Bayes' rule given as $p(y|x) = p(x|y)\,p(y)\,/\,p(x)$ can also be kernelized by

$$\mu_{Y|x} = C_{Y|X}\,\varphi(X) = C_{XY}C_{XX}^{-1}\,\varphi(X) \tag{3.39}$$

$$= \frac{(C_{X|Y}\,C_{YY})^T\,\varphi(X)}{C_{(XX)|Y}\,\mu_Y}. \tag{3.40}$$

4 Generalized Kernel Kalman Filter

In this chapter, the foundations of the KF and RKHS are combined by embedding the update equations of the KF in a RKHS, thereby generalizing the KF to non-linear feature mappings. Furthermore, the sub-space GKKF is introduced which uses only a sub-set of the training samples for representing the state embedding, while using all samples for the model learning. Thus, the operations of the GKKF become also computable for large data sets. At the end of this chapter, the implementation of the sub-space GKKF algorithm is illustrated using pseudocode.

4.1 RKHS Embedding of the Kalman Filter

The formulations of the KF are embedded into a RKHS in order to conduct state estimation for systems with unknown, probably non-linear, system dynamics. As typically in environments where the KF is used, we assume to receive an observation y_t from the system at time point t whose true hidden state is denoted by x_t. The main idea for the embedding is now to represent the state x_t with a mean embedding $\mu_{\varphi(x_t)}$ in a RKHS with the kernel $k(x_t, x_{t+1}) = \langle \varphi(x_t), \varphi(x_{t+1}) \rangle$ and with a covariance embedding $\Sigma_{\varphi(x_t)}$ in a tensor-product RKHS with the kernel $h(x_t, x_{t+1}) = \langle \hat{\varphi}(x_t), \hat{\varphi}(x_{t+1}) \rangle$, whereas $\hat{\varphi}(x_t) = \varphi(x_t) - \mu_{\varphi(x_t)}$ is the centered feature mapping [39]. Moreover, the distribution over the observation y_t is embedded into another RKHS, whereas for easier distinction the feature mapping is denoted by $\phi(y_t)$. Also, the notation $\varphi(x_t)$ and $\varphi(x_{t+1})$ will be shortened to φ_t and φ_{t+1} in the following equations.

Starting by embedding the equations of the **prediction update** in a RKHS, the system matrix $\mathbf{F}$ is represented in the Hilbert space by a conditional operator $C_{\varphi'|\varphi}$ that is able to map the *a posteriori* mean embedding μ_{φ_t} of time point t to the mean embedding of the next state $\mu^-_{\varphi_{t+1}}$. The hyphen denotes that the mean is an *a priori* belief. As described for the traditional KF in Section 3.2.2 the system matrix $\mathbf{F}$ is constructed out of equations that are assumed to describe the system dynamics. However, if they are not known, the model has to be learned from a training data set of size m. The same is true in the presented kernelized KF and thus, $C_{\varphi'|\varphi}$ can be calculated using a sample-based estimator. As described in Section 3.3.3 this can be achieved with

$$C_{\varphi'|\varphi} = \mathbf{\Upsilon}_{x'} \left(\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m\right)^{-1} \mathbf{\Upsilon}_x^T, \tag{4.1}$$

whereas $\mathbf{K}_{xx} = \mathbf{\Upsilon}_x^T \mathbf{\Upsilon}_x$. The matrix $\mathbf{\Upsilon}_x = [\varphi(x_1), \ldots, \varphi(x_m - 1)]$ contains the features mappings of all states and $\mathbf{\Upsilon}_{x'} = [\varphi(x_2), \ldots, \varphi(x_m)]$ the mappings of all subsequent states. Therefore, the conditional operator is able to map the mean embeddings of all states to the next states. The parameter λ_T is used to regularize the Gram matrix before the inversion. The prediction update equations of the traditional KF can then be reformulated in a RKHS as

$$\mu^-_{\varphi_{t+1}} = C_{\varphi'|\varphi}\, \mu_{\varphi_t}, \qquad \Sigma^-_{\varphi_{t+1}} = C_{\varphi'|\varphi}\, \Sigma_{\varphi_t}\, C^T_{\varphi'|\varphi} + \mathbf{\Upsilon}_{x'}\, \mathbf{V}\, \mathbf{\Upsilon}^T_{x'}, \tag{4.2}$$

whereas $\mathbf{\Upsilon}_{x'}\, \mathbf{V}\, \mathbf{\Upsilon}^T_{x'}$ is the covariance of the zero-mean Gaussian noise of the transition model which is also learned from the data set. For the **innovation update** we define an observation operator $C_{\phi|\varphi}$ with $\phi(y_t) = C_{\phi|\varphi}\, \varphi(x_t) + \nu$, where ν is zero-mean Gaussian noise with unit variance. Thus, the operator maps the state embedding to the observation embedding and therefore, represents the GKKF equivalent of the observation matrix $\mathbf{H}$ from the original KF. The observation operator is estimated using the training data with $C_{\phi|\varphi} = \mathbf{\Phi}_y \left(\mathbf{K}_{xx} + \lambda_O \mathbf{I_m}\right)^{-1} \mathbf{\Upsilon}_x^T$, whereas $\mathbf{\Phi}_y = [\phi(y_1), \ldots, \phi(y_m)]$ and λ_O is again a regularization parameter. In the original KF equations the *a priori* mean and covariances are used in the innovation update together with the current observation y_t to calculate the Kalman gains and the *a posteriori* belief over the state. The Hilbert space equivalent of the innovation update equations can be formulated as

$$\mu_{\varphi_t} = \mu^-_{\varphi_t} + \mathcal{Q}_t \left(\phi(y_t) - C_{\phi|\varphi}\, \mu^-_{\varphi_t}\right), \qquad \Sigma_{\varphi_t} = \Sigma^-_{\varphi_t} - \mathcal{Q}_t\, C_{\phi|\varphi}\, \Sigma^-_{\varphi_t}, \tag{4.3}$$

whereas the Kalman gain matrix is calculated by

$$\mathcal{Q}_t = \Sigma^-_{\varphi_t}\, C^T_{\phi|\varphi} \left(C_{\phi|\varphi}\, \Sigma^-_{\varphi_t}\, C^T_{\phi|\varphi} + \kappa \mathbf{I}\right)^{-1}. \tag{4.4}$$

The zero-mean Gaussian noise of the observation model is estimated as $\kappa \mathbf{I}_m$. All calculated means and covariances lie in the Hilbert space and thus for the kernelized KF, an additional step is needed to map the embeddings back to the original state space. For this **reconstruction of the state distribution** another conditional operator $C_{X|\varphi}$ needs to be defined. Equivalent to the already mentioned conditional operators, it is calculated using a sample-based estimator resulting in

$$C_{X|\varphi} = \mathbf{X} (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T, \tag{4.5}$$

whereas $\mathbf{X} = [x_1, \ldots, x_m]$. The reconstruction is now conducted by applying the conditional operator to the mean and covariance embeddings which yields

$$\mu_{x_t} = C_{X|\varphi}\, \mu_{\varphi_t}, \qquad \Sigma_{x_t} = C_{X|\varphi}\, \Sigma_{\varphi_t}\, C_{X|\varphi}^T. \tag{4.6}$$

4.2 Finite-sample RKHS Embedding

The GKKF embeds the state belief in a potentially infinite-dimensional Hilbert space. Thus, μ_{φ_t} and Σ_{φ_t} cannot be computed directly and need to be estimated. The estimation is done by representing the mean embedding at time point t only by a finite vector $m_t \in \mathbb{R}^{mx1}$ and the covariance by a finite-dimensional matrix $S_t \in \mathbb{R}^{mxm}$ through

$$\mu_{\varphi_t} = \boldsymbol{\Upsilon}_{x'}\, m_t, \qquad \Sigma_{\varphi_t} = \boldsymbol{\Upsilon}_{x'}\, S_t\, \boldsymbol{\Upsilon}_{x'}^T. \tag{4.7}$$

When inserted into the Equations 4.2 we receive a **finite-sample prediction update** formulation where the finite-dimensional *a priori* mean embedding is calculated as

$$\begin{aligned}
\mu_{\varphi_{t+1}}^- &= C_{\varphi'|\varphi}\, \mu_{\varphi_t} && (4.8)\\
\boldsymbol{\Upsilon}_{x'}\, m_{t+1}^- &= \boldsymbol{\Upsilon}_{x'} (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'}\, m_t && (4.9)\\
m_{t+1}^- &= (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \mathbf{K}_{xx'} m_t && (4.10)\\
&= \mathbf{T}\, m_t. && (4.11)
\end{aligned}$$

The transition matrix $\mathbf{T} = (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}$ is the finite-dimensional equivalent to the conditional operator $C_{\varphi'|\varphi}$ and forms the learned model of the underlying system's dynamics. For the covariance embedding estimation we receive

$$\begin{aligned}
\Sigma_{\varphi_{t+1}}^- &= C_{\varphi'|\varphi}\, \Sigma_{\varphi_t}\, C_{\varphi'|\varphi}^T + \boldsymbol{\Upsilon}_{x'}\, \mathbf{V}\, \boldsymbol{\Upsilon}_{x'}^T && (4.12)\\
\boldsymbol{\Upsilon}_{x'}\, S_{t+1}^-\, \boldsymbol{\Upsilon}_{x'}^T &= \boldsymbol{\Upsilon}_{x'} (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'}\, S_t\, \boldsymbol{\Upsilon}_{x'}^T \left(\boldsymbol{\Upsilon}_{x'} (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T\right)^T + \boldsymbol{\Upsilon}_{x'}\, \mathbf{V}\, \boldsymbol{\Upsilon}_{x'}^T && (4.13)\\
\boldsymbol{\Upsilon}_{x'}\, S_{t+1}^-\, \boldsymbol{\Upsilon}_{x'}^T &= \boldsymbol{\Upsilon}_{x'} (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'}\, S_t\, \boldsymbol{\Upsilon}_{x'}^T \boldsymbol{\Upsilon}_x (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_{x'}^T + \boldsymbol{\Upsilon}_{x'}\, \mathbf{V}\, \boldsymbol{\Upsilon}_{x'}^T && (4.14)\\
S_{t+1}^- &= (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}\, S_t\, \mathbf{K}_{xx'}^T (\mathbf{K}_{xx} + \lambda_T \mathbf{I}_m)^{-1} + \mathbf{V} && (4.15)\\
&= \mathbf{T}\, S_t\, \mathbf{T}^T + \mathbf{V}. && (4.16)
\end{aligned}$$

As a next step, the equations for a **finite-sample innovation update** are introduced. The finite-dimensional Kalman gain matrix $\mathbf{Q}_t \in \mathbb{R}^{mxm}$ is estimated by

$$\mathbf{Q}_t = S_t^- \mathbf{O}^T (\mathbf{G}_{yy}\, \mathbf{O}\, S_t^- \mathbf{O}^T + \kappa\, \mathbf{I}_m)^{-1}, \tag{4.17}$$

whereas $\mathbf{G}_{yy} = \mathbf{\Phi}_y^T \mathbf{\Phi}_y$ is the Gram matrix of the embedded observations. The learned observation model of the underlying system is given by $\mathbf{GO} = \mathbf{G}_{yy}\mathbf{O}$, whereas $\mathbf{O} = (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}$. The finite-dimensional Kalman gains can be extracted from the infinite-dimensional ones since $\mathcal{Q}_t = \mathbf{\Upsilon}_{x'} \mathbf{Q}_t \mathbf{\Phi}_y^T$. All steps for deriving $\mathbf{Q}_t$ are shown in detail in Appendix A.2.
Applying $\mathbf{Q}_t$, the finite-dimensional *a posteriori* mean embedding can be derived as

$$\mu_{\varphi_t} = \mu_{\varphi_t}^- + \mathcal{Q}_t (\phi(y_t) - C_{\phi|\varphi}\, \mu_{\varphi_t}^-) \tag{4.18}$$
$$\mathbf{\Upsilon}_{x'}\, m_t = \mathbf{\Upsilon}_{x'}\, m_t^- + \mathbf{\Upsilon}_{x'}\, \mathbf{Q}_t \mathbf{\Phi}_y^T \left(\phi(y_t) - \mathbf{\Phi}_y\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{\Upsilon}_x^T\, \mathbf{\Upsilon}_{x'}\, m_t^-\right) \tag{4.19}$$
$$m_t = m_t^- + \mathbf{Q}_t\, (\mathbf{\Phi}_y^T\, \phi(y_t) - \mathbf{\Phi}_y^T\, \mathbf{\Phi}_y\, \mathbf{O}\, m_t^-) \tag{4.20}$$
$$m_t = m_t^- + \mathbf{Q}_t\, (k_{:y_t} - \mathbf{G}_{yy}\, \mathbf{O}\, m_t^-). \tag{4.21}$$

The observations are represented by the kernel vector $k_{:y_t} = [k(y_1, y_t), \ldots, k(y_m, y_t)]$. The finite-dimensional *a posteriori* covariance embedding is then calculated by

$$\Sigma_{\varphi_t} = \Sigma_{\varphi_t}^- - \mathcal{Q}_t\, C_{\phi|\varphi}\, \Sigma_{\varphi_t}^- \tag{4.22}$$
$$\mathbf{\Upsilon}_{x'}\, S_t\, \mathbf{\Upsilon}_{x'}^T = \mathbf{\Upsilon}_{x'}\, S_t^-\, \mathbf{\Upsilon}_{x'}^T - \mathbf{\Upsilon}_{x'}\, \mathbf{Q}_t \mathbf{\Phi}_y^T\, \mathbf{\Phi}_y\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{\Upsilon}_x^T\, \mathbf{\Upsilon}_{x'}\, S_t^-\, \mathbf{\Upsilon}_{x'}^T \tag{4.23}$$
$$S_t = S_t^- - \mathbf{Q}_t\, \mathbf{G}_{yy}\, \mathbf{O}\, S_t^-. \tag{4.24}$$

As for the infinite-dimensional case, the **reconstruction of the state distribution** is needed to map the mean and covariance embeddings back to the original space. For the mean the derivations are given by

$$\mu_{x_t} = C_{X|\varphi}\, \mu_{\varphi_t} \tag{4.25}$$
$$\mu_{x_t} = \mathbf{X}\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{\Upsilon}_x^T\, \mathbf{\Upsilon}_{x'}\, m_t \tag{4.26}$$
$$= \mathbf{X}\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{K}_{xx'}\, m_t \tag{4.27}$$
$$= \mathbf{X}\, \mathbf{O}\, m_t \tag{4.28}$$

and for the covariance by

$$\Sigma_{x_t} = C_{X|\varphi}\, \Sigma_{\varphi_t}\, C_{X|\varphi}^T \tag{4.29}$$
$$= \mathbf{X}\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{\Upsilon}_x^T\, \mathbf{\Upsilon}_{x'}\, S_t\, \mathbf{\Upsilon}_{x'}^T \left(\mathbf{X}\, (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1}\, \mathbf{\Upsilon}_x^T\right)^T \tag{4.30}$$
$$= \mathbf{X}\, \mathbf{O}\, \mathbf{S}_t\, \mathbf{O}^T\, \mathbf{X}^T. \tag{4.31}$$

4.3 Sub-space GKKF

As stated by the authors in [39], the GKKF possesses almost cubic computational complexity for the number of training samples due to the inversion of the Gram matrix $\mathbf{K}_{xx} \in \mathbb{R}^{mxm}$. Thus, for large training sets the calculations become practically intractable. However, a sub-space GKKF variant can be defined that allows to work with large data sets. The core idea is to represent the mean embedding only by a subset of the training samples, whereas still all samples are used to learn the model. To achieve this, a sub-space feature mapping $\mathbf{\Gamma}_x = [\varphi(x_1), \ldots, \varphi(x_{n-1})]$ which contains the mappings of only n training samples is defined, such that $\mathbf{\Gamma}_x \subset \mathbf{\Upsilon}_x$. The Gram matrix is then calculated as $\overline{\mathbf{K}}_{\mathbf{xx}} = \mathbf{\Upsilon}_x^T\, \mathbf{\Gamma}_x$ with dimensions $\overline{\mathbf{K}}_{\mathbf{xx}} \in \mathbb{R}^{mxn}$ leading to new conditional embedding operators for the model learning which are called sub-space conditional embedding operators and are introduced in [55].
The formulations for the **prediction update** of the sub-space GKKF stay the same as for the full-space GKKF with

$$\mu_{\varphi_{t+1}}^- = C_{\varphi'|\varphi}^S\, \mu_{\varphi_t}, \qquad \Sigma_{\varphi_{t+1}}^- = C_{\varphi'|\varphi}^S\, \Sigma_{\varphi_t}\, C_{\varphi'|\varphi}^{ST} + \mathbf{\Upsilon}_{x'}\, \mathbf{V}\, \mathbf{\Upsilon}_{x'}^T, \tag{4.32}$$

but the conditional operator is now given by $C^S_{\varphi'|\varphi} = \boldsymbol{\Upsilon}_{x'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x$, whereas $\mathbf{L}^S_T = (\overline{\mathbf{K}}^T_{xx} \overline{\mathbf{K}}_{xx} + \lambda_T \mathbf{I}_n)^{-1}$.
The same is true for the **innovation update** equations given as

$$\mu_{\varphi_t} = \mu^-_{\varphi_t} + \mathcal{Q}^S_t (\phi(y_t) - C^S_{\phi|\varphi} \mu^-_{\varphi_t}), \qquad \Sigma_{\varphi_t} = \Sigma^-_{\varphi_t} - \mathcal{Q}^S_t C^S_{\phi|\varphi} \Sigma^-_{\varphi_t}. \tag{4.33}$$

Here, a sub-space conditional operator that maps the state embeddings to the observation embeddings is calculated with $C^S_{\phi|\varphi} = \boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}^S_O \boldsymbol{\Gamma}^T_x$, whereas $\mathbf{L}^S_O = (\overline{\mathbf{K}}^T_{xx} \overline{\mathbf{K}}_{xx} + \lambda_O \mathbf{I}_n)^{-1}$. Moreover, a sub-space Kalman gain matrix is needed that is build using the sub-space conditional operator. It can be computed as

$$\mathcal{Q}^S_t = \Sigma^-_{\varphi_t} C^{ST}_{\phi|\varphi} (C^S_{\phi|\varphi} \Sigma^-_{\varphi_t} C^{ST}_{\phi|\varphi} + \kappa \mathbf{I})^{-1}. \tag{4.34}$$

Again, for the **reconstruction of the state distribution** a third conditional operator is needed which is calculated by $C^S_{X|\varphi} = \mathbf{X} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_O \boldsymbol{\Gamma}^T_x$ and utilized to map the feature embeddings back to the original space with the equations

$$\mu_{x_t} = C^S_{X|\varphi} \mu_{\varphi_t}, \qquad \Sigma_{x_t} = C^S_{X|\varphi} \Sigma_{\varphi_t} C^{ST}_{X|\varphi}. \tag{4.35}$$

As for the full-space GKKF, the mean and covariance embeddings are possibly infinite-dimensional and need to be estimated to become directly computable. However, now only a subset of the training samples is used for the state representation given by

$$n_t = \boldsymbol{\Gamma}^T_x \mu_{\varphi_t}, \qquad P_t = \boldsymbol{\Gamma}^T_x \Sigma_{\varphi_t} \boldsymbol{\Gamma}_x, \tag{4.36}$$

whereas $n_t \in \mathbb{R}^{nx1}$ and $P_t \in \mathbb{R}^{nxn}$. With this estimation we can formulate the **finite-sample prediction update** equations for the sub-space GKKF that allows for the computation of a finite-dimensional sub-space *a priori* mean embedding by

$$\mu^-_{\varphi_{t+1}} = C^S_{\varphi'|\varphi} \mu_{\varphi_t} \tag{4.37}$$
$$\boldsymbol{\Gamma}^T_x \mu^-_{\varphi_{t+1}} = \boldsymbol{\Gamma}^T_x \boldsymbol{\Upsilon}_{x'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x \mu_{\varphi_t} \tag{4.38}$$
$$n^-_{t+1} = \overline{\mathbf{K}}^T_{xx'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \, n_t \tag{4.39}$$
$$n^-_{t+1} = \mathbf{T} n_t, \tag{4.40}$$

whereas $\mathbf{T}^S = \overline{\mathbf{K}}^T_{xx'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T$ is the sub-space transition matrix. For the covariance embedding estimation we end up at

$$\Sigma^-_{\varphi_{t+1}} = C^S_{\varphi'|\varphi} \Sigma_{\varphi_t} C^{ST}_{\varphi'|\varphi} + \boldsymbol{\Upsilon}_{x'} \mathbf{V} \boldsymbol{\Upsilon}^T_{x'} \tag{4.41}$$
$$\Sigma^-_{\varphi_{t+1}} = \boldsymbol{\Upsilon}_{x'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x \Sigma_{\varphi_t} \left(\boldsymbol{\Upsilon}_{x'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x \right)^T + \boldsymbol{\Upsilon}_{x'} \mathbf{V} \boldsymbol{\Upsilon}^T_{x'} \tag{4.42}$$
$$\Sigma^-_{\varphi_{t+1}} = \boldsymbol{\Upsilon}_{x'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x \Sigma_{\varphi_t} \boldsymbol{\Gamma}_x \mathbf{L}^S_T \overline{\mathbf{K}}^T_{xx} \boldsymbol{\Upsilon}^T_{x'} + \boldsymbol{\Upsilon}_{x'} \mathbf{V} \boldsymbol{\Upsilon}^T_{x'} \tag{4.43}$$
$$\boldsymbol{\Gamma}^T_x \Sigma^-_{\varphi_{t+1}} \boldsymbol{\Gamma}_x = \overline{\mathbf{K}}^T_{xx'} \overline{\mathbf{K}}_{xx} \mathbf{L}^S_T \boldsymbol{\Gamma}^T_x \Sigma_{\varphi_t} \boldsymbol{\Gamma}_x \mathbf{L}^S_T \overline{\mathbf{K}}^T_{xx} \overline{\mathbf{K}}_{xx'} + \boldsymbol{\Gamma}^T_x \boldsymbol{\Upsilon}_{x'} \mathbf{V} \boldsymbol{\Upsilon}^T_{x'} \boldsymbol{\Gamma}_x \tag{4.44}$$
$$P^-_{t+1} = \mathbf{T}^S P_t \mathbf{T}^{ST} + \boldsymbol{\Gamma}^T_x \boldsymbol{\Upsilon}_{x'} \mathbf{V} \boldsymbol{\Upsilon}^T_{x'} \boldsymbol{\Gamma}_x. \tag{4.45}$$

For the **finite-sample innovation update** we start by defining a finite-dimensional sub-space Kalman gain matrix $\mathbf{Q}^S_t \in \mathbb{R}^{nxn}$ which is estimated by $\mathbf{Q}^S_t = P^-_t \mathbf{L}^S_O (\overline{\mathbf{K}}^T_{xx} \mathbf{G}_{\mathbf{yy}} \mathbf{O}^S P^-_t \mathbf{L}^S_O + \kappa \mathbf{I}_n)^{-1} \overline{\mathbf{K}}^T_{xx}$, whereas $\mathbf{G}\mathbf{O}^S = \mathbf{G}_{\mathbf{yy}} \mathbf{O}^S$ with $\mathbf{O}^S = \overline{\mathbf{K}}_{xx} \mathbf{L}^S_O$ is the observation matrix. The relation to the infinite-dimensional sub-space Kalman gain matrix is given by $\boldsymbol{\Gamma}^T_x \mathcal{Q}^S_t = \mathbf{Q}^S_t \boldsymbol{\Phi}^T_y$.

The complete derivation steps are shown in Appendix A.3. The equations for the finite-dimensional sub-space *a posteriori* mean embedding are then given by

$$\mu_{\varphi_t} = \mu^-_{\varphi_t} + \mathcal{Q}^S_t\,(\phi(y_t) - C^S_{\phi|\varphi}\,\mu^-_{\varphi_t}) \tag{4.46}$$

$$\mathbf{\Gamma}^T_x\,\mu_{\varphi_t} = \mathbf{\Gamma}^T_x\,\mu^-_{\varphi_t} + \mathbf{\Gamma}^T_x\,\mathcal{Q}^S_t\,(\phi(y_t) - \mathbf{\Phi}_y\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,\mathbf{\Gamma}^T_x\,\mu^-_{\varphi_t}) \tag{4.47}$$

$$n_t = n^-_t + \mathbf{Q}^S_t\,\mathbf{\Phi}^T_y\,(\phi(y_t) - \mathbf{\Phi}_y\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,n^-_t) \tag{4.48}$$

$$= n^-_t + \mathbf{Q}^S_t\,(\mathbf{\Phi}^T_y\,\phi(y_t) - \mathbf{\Phi}^T_y\,\mathbf{\Phi}_y\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,n^-_t) \tag{4.49}$$

$$= n^-_t + \mathbf{Q}^S_t\,(k_{:y_t} - \mathbf{G}_{yy}\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,n^-_t). \tag{4.50}$$

The finite-dimensional sub-space *a posteriori* covariance embedding is calculated by

$$\Sigma_{\varphi_t} = \Sigma^-_{\varphi_t} - \mathcal{Q}^S_t\,C^S_{\phi|\varphi}\,\Sigma^-_{\varphi_t} \tag{4.51}$$

$$\mathbf{\Gamma}^T_x\,\Sigma_{\varphi_t}\,\mathbf{\Gamma}_x = \mathbf{\Gamma}^T_x\,\Sigma^-_{\varphi_t}\,\mathbf{\Gamma}_x - \mathbf{\Gamma}^T_x\,\mathcal{Q}^S_t\,\mathbf{\Phi}_y\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,\mathbf{\Gamma}^T_x\Sigma^-_{\varphi_t}\,\mathbf{\Gamma}_x \tag{4.52}$$

$$P_t = P^-_t - \mathbf{Q}^S_t\,\mathbf{\Phi}^T_y\,\mathbf{\Phi}_y\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,P^-_t \tag{4.53}$$

$$= P^-_t - \mathbf{Q}^S_t\,\mathbf{G}_{yy}\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,P^-_t. \tag{4.54}$$

The last step is again the **reconstruction of the state distribution**. The derivation for the mean is given by

$$\mu_{x_t} = C^S_{X|\varphi}\,\mu_{\varphi_t} \tag{4.55}$$

$$\mu_{x_t} = \mathbf{X}\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,\mathbf{\Gamma}^T_x\mu_{\varphi_t} \tag{4.56}$$

$$= \mathbf{X}\,\mathbf{O}^S\,n_t \tag{4.57}$$

and for the covariance by

$$\Sigma_{x_t} = C^S_{X|\varphi}\,\Sigma_{\varphi_t}\,C^{ST}_{X|\varphi} \tag{4.58}$$

$$= \mathbf{X}\,\overline{\mathbf{K}}_{xx}\,\mathbf{L}^S_O\,\mathbf{\Gamma}^T_x\Sigma_{\varphi_t}\,\mathbf{\Gamma}_x\,\mathbf{L}^S_O\,\overline{\mathbf{K}}^T_{xx}\,\mathbf{X}^T \tag{4.59}$$

$$= \mathbf{X}\,\mathbf{O}^S\,P_t\,\mathbf{X}\,\mathbf{O}^{ST}. \tag{4.60}$$

4.4 GKKF Algorithm

During this master thesis, both the original GKKF and its sub-space variant were implemented using the Python programming language. However, for the traffic prediction experiments described in Chapter 6, mainly the sub-space GKKF was used in order to make the computations tractable even when using a high number of training samples. The implemented sub-space GKKF is shown as pseudocode in Algorithm 1.

When using the sub-space GKKF for state estimation, the first step is to set different settings for the execution of the algorithm. This includes choosing a training set $data_{train}$, a test set $data_{test}$ and setting the values of multiple hyperparameters that are needed for the calculations and whose values are found through optimization. The hyperparameters λ_T and λ_O are used for regularizing the Gram matrix products before their inversion, whereas the first parameter is used for the transition model during the prediction update step and the later for the observation model during the innovation update step. The bandwidth scaling factors $state_{bw}$ and obs_{bw} decide with which values the bandwidths of the kernel functions, used for the state and observation embeddings, are multiplied with. The bandwidths are found by choosing a subset of training samples and calculating the median of the squared distances between them. The last hyperparameter is the observation noise covariance κ.

In the *preprocess data* function the training and test sets are read and prepared to be used with the GKKF algorithm. Depending on the type and structure of the used data sets different preprocessing steps might be needed. In all cases of transition and observation model learning described in this chapter, it was assumed that the training samples in fact represent the true state x of the system. However, in some cases we might just have access to partial observations y. In such a case we can still make use of the GKKF algorithm when forming an internal state representation by using a window of k observations $y_{t-k+1:t}$, as shown in [56]. However, the formed observation windows are not a fully valid representation of the system state from a theoretical point of view since the observations contain additive noise and were therefore not generated by a Markov process. When no hidden states of the system are given in the training samples, heuristic approaches like the Baum-Welch algorithm are often used to learn the model. Therefore, the authors of [39] propose to combine the Baum-Welch algorithm with the update equations of the GKKF to learn the unknown dynamics of the system. Besides forming a state representation out of observations, the *preprocess data* function could include conducting a Principal Component Analysis (PCA) on the training set to reduce the dimensions of the data sets before using them with the GKKF.

Algorithm 1 Sub-space Generalized Kernel Kalman Filter

settings:
training set $data_{train}$
test set $data_{test}$
regularization parameters λ_T, λ_O
bandwidth scaling factors $state_{bw}$, obs_{bw}
observation noise covariance κ

function PREPROCESS DATA
form state windows (optional)
conduct PCA on $data_{train}$ (optional)

function MODEL LEARNING
Gram matrices $\overline{\mathbf{K}}_{xx}$, $\overline{\mathbf{K}}_{xx'}$, $\mathbf{G}_{yy}$
state matrix $\mathbf{X}$,
transition model matrix $\mathbf{T}^S$
observation model matrix $\mathbf{GO}^S$
initial belief embedding n_1^-, P_1^-

function PROJECTION
while $project$ **do**
Kalman gain $\mathbf{Q}_t^S$
a posteriori covariance embedding $\mathbf{P}_t$
a priori covariance embedding $\mathbf{P}_{t+1}^-$

function TEST MODEL
while $predict$ **do**
if new observation y_t **then**
innovation update:
a posteriori mean embedding n_t

prediction update:
a priori mean embedding n_{t+1}^-

project into state space:
mean prediction μ_{x_t} and covariance prediction Σ_{x_t}

In the *model learning* function the Gram matrices of the state embeddings denoted $\overline{\mathbf{K}}_{xx}$ and $\overline{\mathbf{K}}_{xx'}$ are formed together with the observation Gram matrix $\mathbf{G}_{yy}$. The matrices are then used to estimate the sub-space transition model matrix $\mathbf{T}^S$ and the sub-space observation model matrix $\mathbf{GO}^S$. Moreover, the state matrix $\mathbf{X}$ needed for the projection back to the state space is formed and the initial values for the *a priori* sub-space mean embedding n_1^- and covariance embedding P_1^- are set.

The GKKF algorithm can either be used online or offline. When used online, single observations would be omitted from the underlying system and directly used in the innovation update step. However, during the traffic prediction experiments described in Chapter 6 the algorithm was used offline, so that a complete test set $data_{test}$ was already present when testing the model. In the equations for the prediction and innovation updates shown in this chapter, the Kalman gain matrices and the covariances were calculated together with the mean embeddings. However, none of the components depend on the current observation y_t and therefore, they can be calculated before receiving any observation. Thus, the calculation can be done in the function *projection* before testing the model which reduces the computation time needed for the GKKF algorithm. For the online case, the computation of the innovation and prediction update steps becomes much faster which is important when the observations are received directly from the system. In the offline mode, the projection of $\mathbf{Q}_t^S$, P_t and P_{t+1}^- allows to save computation time because when testing the model simultaneously with multiple test sets the components only have to be calculated once.

In the last step of the algorithm the model is tested, whereas the only step left in the innovation and prediction update is the calculation of the *a posteriori* and the *a priori* sub-space mean embeddings n_t and n_{t+1}^-, respectively. After the prediction, the mean and covariance embeddings are projected back into the state space. The prediction and innovation update steps can be executed alternately. However, if the observations arrive irregularly only every p-th time step, the algorithm will solely execute the prediction update step and thus perform a p-step prediction of the system state.

5 Software-Defined Network Experiment Framework

One goal during this master thesis was to create a tool that allows the user to set up a virtualized SDN and perform experiments with it. The result is the SDN experiment framework described in this chapter. It was developed using the Python programming language with the aim to execute custom experiments in custom topologies on a scalable testbed. This chapter presents the framework by introducing the different components combined in the framework, followed by a description of the steps included when starting, running and evaluating a network experiment with it.

5.1 Components

In order to build a SDN experiment framework that is scalable, multiple existing software components were combined. The first component is the Distributed Internet Traffic Generator (D-ITG) from Section 5.1.1 which is used to generate traffic for the network experiments. The SDN itself is emulated by Mininet described in Section 5.1.3 which mainly relies on the Open vSwitch (OVS), introduced in Section 5.1.2, as its supported OpenFlow enabled virtual switch implementation. To make the experiments scalable, the Mininet network is split using MaxiNet from Section 5.1.4 and executed on a set of Virtual Machines (VMs) which in turn are provided by the ToMaTo testbed described in Section 5.1.5.

5.1.1 Distributed Internet Traffic Generator

One important component of the experiment framework is the traffic generator which is used to generate network traffic flows in the emulated network. In the network experiments, single flow traces collected in real networks should be replayed as they occurred in the real network which means that the packet timing and packet sizes should replicate the real scenario as exact as possible.
There are several traffic generators that support such a trace-based traffic generation like TCPivo [57], TCPreplay [58] or TCPopera [59]. However, because of the possibility to schedule multiple flows and the more comprehensive and convenient way of altering properties of single flows, the traffic generator D-ITG was selected to be integrated into the experiment framework. Furthermore, D-ITG offers more advanced features for calculating and logging network metrics and provides also an analytical model-based traffic generation mode. In this mode D-ITG is capable of producing realistic network workloads that replicate stochastic processes [60]. During the experiments, this functionality was not utilized but might be useful in some other scenarios which is why the experiment framework becomes more flexible by integrating the D-ITG traffic generator rather than a trace-based only generator.

An architecture overview of D-ITG is shown in Figure 5.1. The core features are provided by *ITGSend* and *ITGRecv*. *ITGSend* started on the sender host can run in single-flow or multi-flow mode to send flows to one or multiple different receiver hosts running *ITGRecv*. When using the multi-flow mode a new thread is spawned for every flow. Besides, a signaling channel is established between each couple of *ITGSend* and *ITGRecv* processes to control the traffic flows between them. When *ITGSend* is used for trace-based traffic generation a flow is represented by a pair of files that contain the packet size and Inter Departure Time (IDT) information for every data packet of the flow. *ITGSend* reads those files and replays the flow as accurate as possible. The third possible *ITGSend* mode is the daemon mode. In this mode *ITGSend* is remotely controlled over the D-ITG Application Programming Interface (API). An example program using the interface is denoted as *ITGManager* in Figure 5.1. A similar mode is also existing for *ITGRecv* where it can be remotely configured and controlled from a *ITGSend* component.
Both *ITGSend* and *ITGRecv* can log information about every data packet they sent or received. The log files are either created locally or sent to the *ITGLog* component located on a remote machine. After the traffic generation is finished, the *ITGDec* component can be used to analyze the log files and to calculate flow statistics and QoS metrics like bitrate, packet loss, delay or jitter. In a subsequent step, the provided Octave script *ITGplot* can be used to plot the results.

When producing trace-based traffic, D-ITG can only send unidirectional flows between a sender and receiver. To simulate a bidirectional communication of application-level protocols, the flows need to be split into request and response parts and replayed using a *ITGSend* and *ITGRecv* instance on both the client and server side. However, as with other traffic generators, this approach is not capable of producing a truly synchronized bidirectional communication as needed for TCP. When the main goal is to reproduce the characteristic flow patterns on the emulated links this is also not necessary. Though, in other scenarios this might be needed. The traffic generation platform described in [60], which

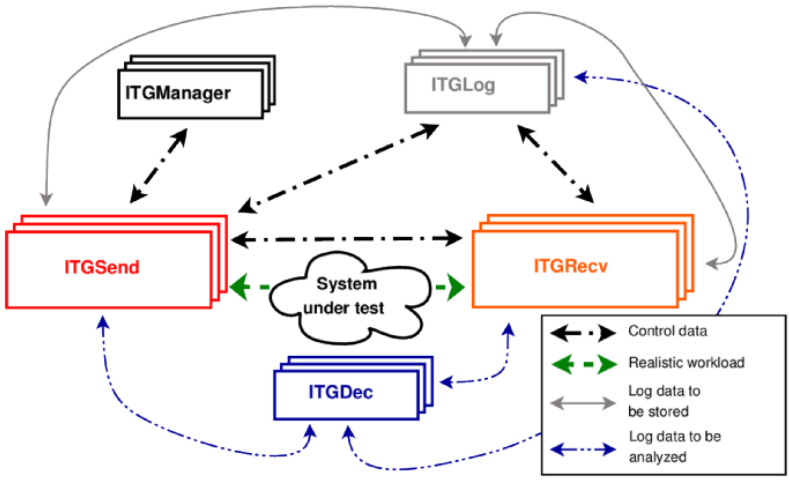


Figure 5.1.: Architecture of the Distributed Internet Traffic Generator [2]

is based on D-ITG, is said to support synchronized bidirectional flow generation. However, this feature is not included in the open-source publicly available D-ITG variant. Therefore, D-ITG was extended during this master thesis by a simple mechanism illustrated in Figure 5.2 that allows for producing synchronized bidirectional TCP flows.
Given an application-level flow, one host is denoted as the client and the other host as the server. Every host runs an *ITGSend* and *ITGRecv* instance in parallel and owns a set of files for every flow communication it maintains to another host. The files *flow_idt* and *flow_pktz* contain the IDT and packet size information for every data packet of the dedicated flow. However, the files contain only the packets associated with the client or server side. The *send_info* files contain tuples showing how many data packets the host has to receive before sending its own packets. The files *send_idt* and *send_pktz* build a sending queue that contains the IDT and packet size information for packets that still need to be send by the host. When the generation process for a flow is started the *send_idt* and *send_pktz* files are empty at the server side and contain the information for at least one packet at the client side. The *ITGSend* process is constantly checking both files for updates. Thus, the *ITGSend* client instance will read the packet information and begin sending packets to the server. The *ITGRecv* instance on the server side receives the packets, logs them as usual and validates, using the *send_info* file, if it has to send a response in return. If not, it waits for more packets to come. If it has to send a response, the information for those packets is copied from the *flow_idt* and *flow_pktz* files to the *send_idt* and *send_pktz* files. The *ITGSend* instance on the server side will notice the new file content and send the appropriate packets. The same process is now happening alternately on the client and server side until the flow is finished. Since a host is always waiting for all data packets as they occurred in the real flow before sending its own packets, this approach can only work for TCP traffic since packets might get lost when using UDP.

5.1.2 Open vSwitch

Open vSwitch (OVS) is a multi-layer, open-source virtual switch that supports many hypervisor platforms like Kernel-based Virtual Machine (KVM) [61], Xen [62] or VirtualBox [63]. In virtual networks OVS switches act as the provider of network services for VMs which they interconnect. Since OVS was, in contrast to other virtual switch implementations, designed for flexibility and general-purpose usage, it supported OpenFlow since the beginning because it makes the switch easily re-programmable [3]. Its current version 2.5.0 supports OpenFlow up to version 1.3.
The main components of OVS are shown in Figure 5.3. It consists of the processes *ovsdb-server* and *ovs-vswitchd* in the userspace and the kernel datapath module in the kernel space, whereas the later two are involved in the data packet forwarding. Whenever a packet type is received for the first time from the virtual switch, it is directed to *ovs-vswitchd*. Here, it is decided how the packet should be handled making use of the different OpenFlow specific tables explained in Section 3.1.2 that were populated by the controller. The data packets are then given back to the kernel datapath module together with an action that is executed and cached. Thus, a call to the userspace can be omitted for similar subsequent packets. In this kernel version OVS does only support virtualization technologies were the kernel can be modified or extended by modules. OVS can also operate entirely in userspace without using a kernel module. However, this reduces the performance of the switch and is at the moment declared as an experimental feature [64].
The third part of OVS is the *ovsdb-server*. The controller uses the management protocol Open vSwitch Database Management Protocol (OVSDB) to connect to the *ovsdb-server* process and to configure the OVS. In contrast to OpenFlow, OVSDB is used to set more stable settings of the OVS, e.g. to add a port to the switch or to remove it.

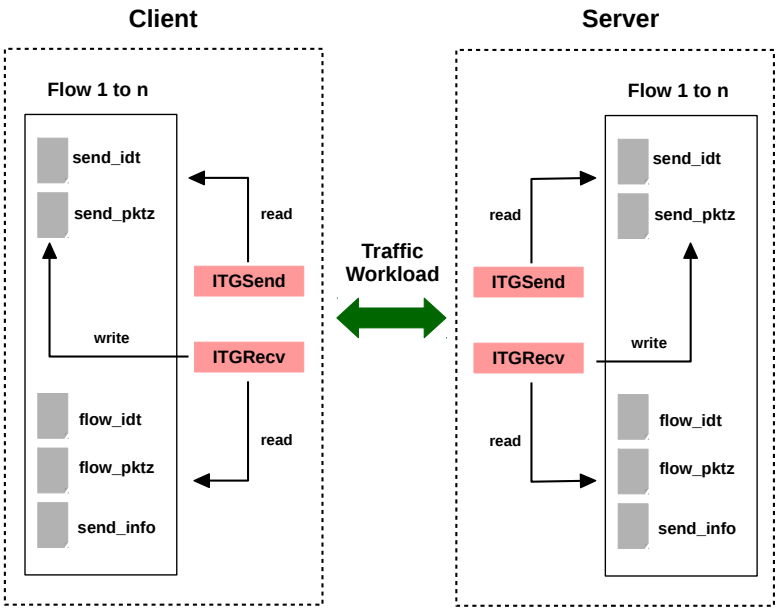


Figure 5.2.: Synchronized Bidirectional Flow Generation with Custom D-ITG

5.1.3 Mininet

Mininet forms the basis around which the experiment framework was built. Mininet is an network emulator that allows to rapidly prototype realistic virtual networks consisting of switches, hosts, links and controllers on a single machine [4]. In contrast to the commercial network emulator EstiNet [65], Mininet is open-source. Moreover, compared to network simulators like ns-3 [66] or OMNeT++ [67], Mininet behaves more realistic because existing applications and network stacks are used without modifications. Mininet allows for the deployment of custom topologies and the development of applications implementing new network services or functionalities which makes Mininet a flexible network testbed. Because of the realistic behavior of its elements, prototypes developed in Mininet can be deployed to hardware-based systems with minor modifications. Furthermore, Mininet networks are interactive which means they can be managed by the user in real-time.

Mininet uses a light-weight virtualization technique called process-based virtualization for creating a virtualized network. Every host and switch of the network runs in its own process but all of them lie in the same process namespace, whereas a process namespace is a container for the process state. Therefore, every node sees the same processes and directories and has access to the same applications installed on the underlying Operating System (OS). However, Mininet uses a Linux feature to embed the nodes in different network namespaces as illustrated in Figure 5.4. Every host is located in its own network namespace so that separate network interfaces, routing tables and Address Resolution Protocol (ARP) tables can be defined for every host. All switches reside in the root namespace. The links connecting the switches and hosts are realized with virtual Ethernet pairs and link parameters like bandwidth, delay or packet loss can be set by the user. Mininet supports virtual switches like Linux bridge but also user and kernel space OpenFlow enabled switches like OVS, whereas the current Mininet version 2.2.1 supports the newest OVS version 2.5.0. Therefore, Mininet is especially useful for developing and testing applications that should run in a SDN.
As shown in Figure 5.4 the controller connects over TCP to the switches, whereas it can run on the same machine or on a different machine when declared as a remote controller. As the controller, the user can either use one already included in Mininet that implements a simple Ethernet learning switch or use any other SDN controller like Ryu [68], POX [69] or Floodlight [70] together with custom developed applications. Ryu was selected as the SDN controller implementation integrated into the experiment framework because it supports the newest OpenFlow version 1.5.0 and like Mininet it is written in Python and provides a Python API.

Because of its flexibility, its ability to run with unmodified code, its support for OpenFlow enabled switches and easy-to-use Python API, Mininet appeared as the most promising network emulator to serve as the basis for the experiment framework. However, as mentioned, all Mininet components run on the same system which leads to resource limitations because the resources need to be shared among the virtual hosts and switches. Besides, depending on the machine Mininet is running on, other processes might also demand a share of the computation resources which can make the

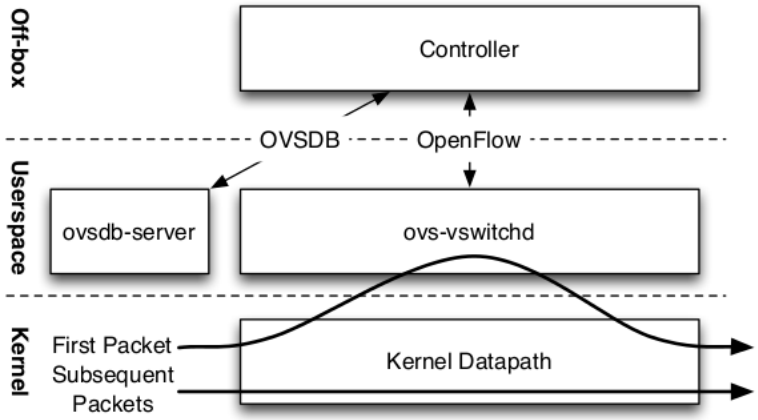


Figure 5.3.: Components and Interfaces of an Open vSwitch [3]

experiments less repeatable. Therefore, besides using the experiment framework together with Mininet locally, it should allow the user to split the Mininet network between multiple machines so that it becomes more scalable. This was achieved by using MaxiNet described in the next section.

5.1.4 MaxiNet

MaxiNet is an extension to Mininet that can be used to span an emulated network over multiple physical machines [5]. In its current stable version 1.0.0 it supports Mininet version 2.2.1. An overview over the MaxiNet architecture is shown in Figure 5.5. Given an computing environment consisting of multiple physical machines in which MaxiNet is used, one machine is declared as the Frontend, whereas the rest is denoted as Workers. The Frontend machine is the central entity used to configure and control the Mininet network by using th MaxiNet API. Internally, MaxiNet utilizes the Mininet API to split the desired Mininet network specified by the user between the different workers, whereas also the Frontend can take a share of the network if needed. On every Worker an unmodified Mininet instance is started, but by connecting the workers over Generic Routing Encapsulation (GRE) tunnels the different instances act as one logical Mininet network. The user can either decide by himself which share of the Mininet network runs on which Worker or he can let MaxiNet make the partitioning decision automatically using the graph partitioning library METIS [71]. After the network is set up, the user can interact with it over the MaxiNet Command-Line Interface (CLI) or API, almost as if he would control just one local Mininet instance.

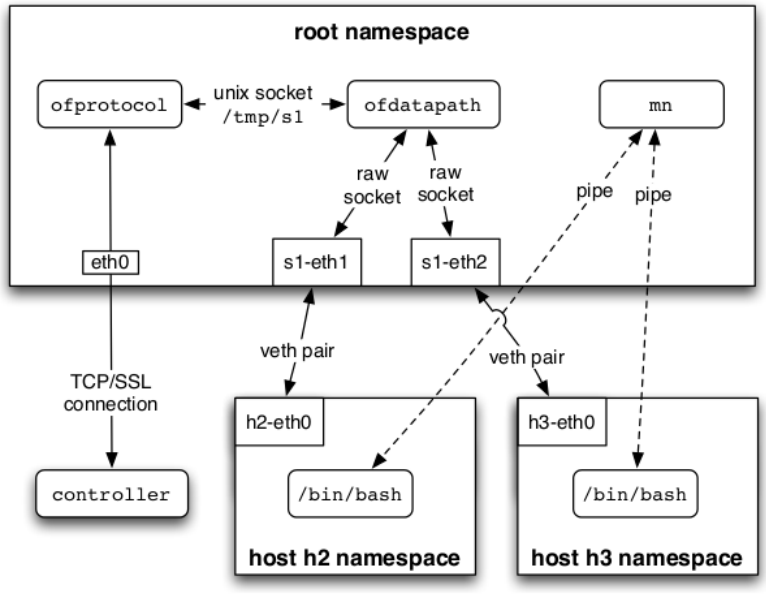


Figure 5.4.: Network Namespaces of a Virtualized Mininet Network [4]

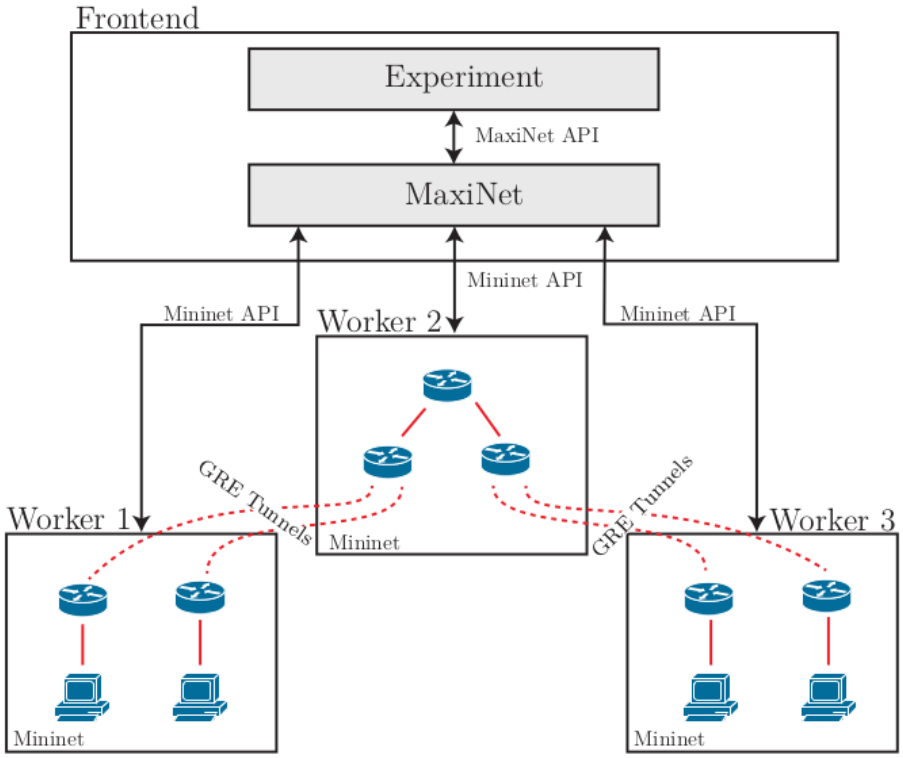


Figure 5.5.: MaxiNet Architecture [5]

5.1.5 ToMaTo

ToMaTo stands for Topology Management Tool and is a topology-centric network testbed that is run by the ToMaTo consortium, an affiliation of universities and research institutes from Europe and North America. Every member contributes to the project by providing computation resources.
In comparison to other experimental facility software solutions, ToMaTo offers a unique combination of features [72]. A user can create virtual network topologies for their experiments that span globally over the different sites in Europe and North America. A topology can be set up using a web-based editor shown in Figure 5.6 or via the Python API of the back-end. ToMaTo offers two different virtual machine technologies that are used to emulate the devices of the network, namely OpenVZ [73] and KVM. OpenVZ is a container-based virtualization solution, whereas KVM offers full virtualization by emulating all hardware needed by the guest OS. Therefore, KVM brings maximal flexibility in choosing and configuring the VMs, but it also comes at a higher cost regarding memory usage and performance reduction. When deciding for OpenVZ the user can select between pre-configured templates for Debian, Ubuntu or set up the devices as Ryu OpenFlow controller or Floodlight OpenFlow controller. For KVM the user can select between images of Debian, Ubuntu or a pre-configured OpenFlow switch. In addition to using pre-built VMs, users can also upload their own custom images. Every VM can be accessed over Virtual Network Computing (VNC), either using client software or directly over a web-based console. Besides, ToMaTo offers network elements like switches and hubs to build the desired topology. The links between the VMs are called connectors and are also emulated. This allows the user to set different link properties like latency or bandwidth. Furthermore, packets transmitted over the links can automatically be captured and downloaded for later analysis. ToMaTo also offers basic monitoring functionality that enables the user to observe the resource usage of their topologies during experiments.

The ToMaTo testbed provides the underlying computation resources for the experiment framework if the experiments should not be run on a local machine. Because of its flexibility in setting up custom network topologies consisting of VMs and the possibility of configuring every node in the network, ToMaTo can be used together with Mininet and MaxiNet to form a highly scalable testbed for virtualized SDNs. With this constellation the Mininet network is not necessarily split between multiple physical machines, as it is usually done when used with MaxiNet, but between multiple VMs which can however be located on different physical machines.

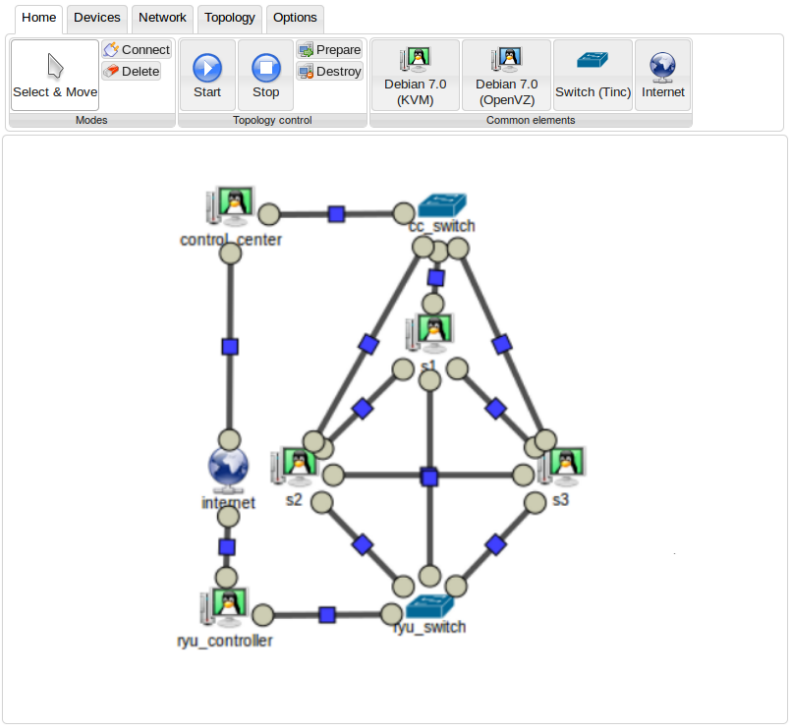


Figure 5.6.: ToMaTo Web-based Editor

5.2 Workflow of the Experiment Framework

The experiment framework allows the user to conduct network experiments either on a single local machine or on multiple VMs by making use of the ToMaTo testbed. The workflow shown in Figure 5.7 illustrates the involved steps when using ToMaTo.

In the first step of the workflow, the user has to define the topology he wants to conduct experiments on. The framework contains examples for the fat-tree topology variant shown in [74] and a custom topology consisting of six switches connected in an arrangement that resembles a sand-glass and is therefore called sand-glass topology. Overall, ten hosts are connected to the switches in the sand-glass topology. Besides the provided topologies, the user can work with every other topology when implementing an appropriate Python class in the framework. Additionally, a JavaScript Object Notation (JSON) file is needed for every topology which defines the network of VMs that is created on the ToMaTo testbed and later used to emulate the SDN. During this master thesis, the desired topology was always exactly rebuilt on ToMaTo so that every switch of the network runs on an own VM. However, depending on the size of the network and the user preferences this could also be handled differently. Thus, it would also be possible to run all switches on a single VM. The needed JSON file can either be created by setting up the topology once on the ToMaTo testbed by hand using the web-based editor and downloading the resulting JSON file, or it can be generated directly from the Python class. The experiment framework can then utilize the ToMaTo API to automatically create a topology on the testbed according to its JSON file.

As shown for a small example topology in Figure 5.6, the created topology on the ToMaTo testbed does not only consist of the switch nodes (s1, s2 and s3), but also of a Ryu controller node and a control center node. The Ryu controller node is connected to all switch nodes and will run the Ryu SDN controller when the network is emulated. The control center does not host parts of the emulated network but the MaxiNet Frontend and is therefore needed to orchestrate the distributed Mininet instances. Both, the control center and the Ryu controller node are connected to the Internet making it easy to upload the network experiment results to a remote repository or to pull an updated version of the experiment framework to the VMs.

All the nodes in the network use KVM as the virtualization technique and Ubuntu 14.04 as the disk image. Since a network could consist of many switches, using OpenVZ for the switch nodes instead might appear beneficial for saving resources. Nevertheless, KVM is used because as described in Section 5.1.2, OVS needs an underlying virtualization technique that allows for kernel modifications to run efficiently which is not provided by OpenVZ. As described in Section 5.1.5 ToMaTo offers also pre-configured images for a Ryu controller or OVS. However, since a lot of configuration has to be done on the OSs of the VMs to make them work with the experiment framework, naked Ubuntu 14.04 images were used. After setting up the topology on the ToMaTo testbed the experiment framework adjusts the OSs by uploading

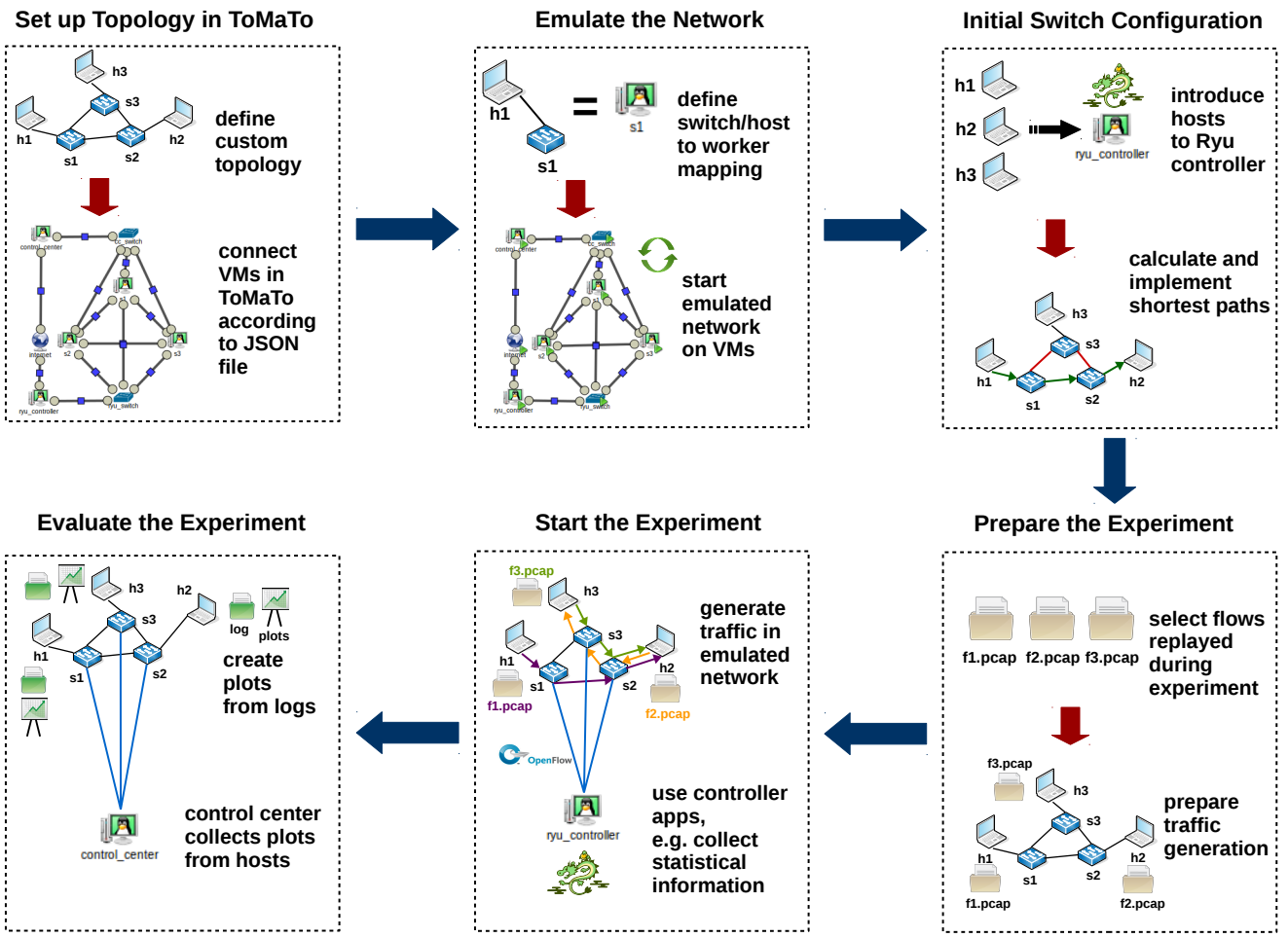


Figure 5.7.: Workflow of the Software-Defined Network Experiment Framework

configuration files and scripts to the VMs that get automatically executed. The configuration steps include installing software like MaxiNet, Mininet, D-ITG and Ryu on the appropriate VMs and configuring the network interfaces. After the installation process is finished, the first step of the workflow is completed and the ToMaTo topology ready to be used.

Before a SDN is emulated on the connected VMs, the framework is used to start the Ryu SDN controller together with two custom applications, the *network app* and the *monitor app*. However, depending on the conducted experiments, other applications would need to be implemented by the user and integrated into the framework. After starting the controller the user needs to define the MaxiNet mapping that decides which hosts and switches of the emulated network should run on which VM (MaxiNet Worker). Since it is not possible with MaxiNet to run also the hosts on separate VMs, they are grouped with the switch they are connected to and hosted together on a single VM. Again, this could be handled differently, but the mapping should fit the decisions made when setting up the VM network on the ToMaTo testbed. After the user decided for a mapping, the framework starts the network emulation by making use of MaxiNet to create a Mininet network and spreads it between the different VMs.

At the beginning of the third workflow step, the emulated network is up and running. However, the ARP tables of the hosts are still empty. The flow tables of the switches contain only one fallback flow entry which sends all data packets that match no other entry to the SDN controller. Those entries are installed by the *network app* during the start up of the emulated network when receiving messages from the switches that are sent to announce their existence. To make the hosts introduce themselves to the controller, the experiment framework forces them to ping other hosts in the network. This leads to the transmission of ARP requests that are forwarded from the switches to the controller. The *network app* observes those requests and uses them to build fake ARP responses that are sent back to the hosts, thereby filling their ARP tables. Furthermore, the *network app* creates an internal graph of all the nodes in the emulated network and calculates the shortest paths between all hosts. Then, depending on the shortest paths found, the flow tables of all switches are populated. As a result, the hosts can now communicate, which is tested by the framework by running a *ping all*

command in the emulated network that makes all hosts ping each other.

In the next workflow step the experiment which should be run on the emulated network is prepared. Here, an experiment is defined as the combination of traffic that is generated in the emulated network and the custom apps that run on the SDN controller during the traffic generation and interact with the switches is some way. The traffic flows are selected and scheduled by the user by defining the sender and target hosts of the flows and the delays between them. Moreover, other flow properties provided by the D-ITG like the flow duration or the number of flow packets can be configured by the user. Additional implementations could also produce a randomized selection of flows for every host. For replaying a flow it simply needs to be provided as a *.pcap* file by the user. The experiment framework is then able to prepare the traffic generation which includes reading in the *.pcap* files, saving them in the appropriate form at the different hosts and creating additional flow dependent files needed by the D-ITG. When the user wants to send the flows through a synchronized bidirectional communication, the framework e.g. has to create the different files shown in Figure 5.2 which are needed for the custom D-ITG variant.
During this master thesis, traffic flows were replayed in real time, thus achieving the same traffic load on the emulated links as it was also present in the real network in which the flows were recorded. However, when the networks are very large and multiple switches need to run on a single VM, or if the link bandwidth and traffic volume is very high, replaying flows in real time can become unfeasible. In such a scenario, network and traffic emulation is still possible when using an approach called time dilation [75]. The idea is to make the OS and its applications believe that time is passing at a lower speed than the actual physical time. The ratio between both perceptions of time is called the time dilation factor. When using a factor of 10 e.g., data packets that arrive at a physical rate of 1 Gbit/s appear to the OS as if they would arrive with 10 Gbit/s. This allows the user to study the behavior of the system at traffic speeds and capacities that could otherwise not be produced with the available hardware. However, a dilation factor increases the runtime of an experiment. Thus, generating a traffic flow with a duration of 1 minute would take 10 minutes when using a dilation factor of 10.

The traffic generation of the experiment is initiated in the next workflow step by starting the D-ITG *ITGSend* and *ITGRecv* processes on the hosts. During the generation, custom controller apps are used to interact with the emulated network. The already mentioned *monitor app* was developed to periodically collect statistical information from the switches at a certain rate. The information received from the switches include counters for the flow entries of the switch flow tables as shown in Appendix A.1. From those counters the *monitor app* is able to determine how much traffic each switch received from every connected host over a certain protocol. This is possible because the *network app* populated the flow tables with entries that allow to split the incoming traffic depending on the destination TCP or UDP port of the data packet. It is also possible to collect those counters for every single flow that is transmitted through the switch if the flow entries are set accordingly. In the end, the *monitor app* can produce a file that contains the kbit values of single flows or of different protocols for every host or switch of the network at a selected rate. Files produced in this manner were planned to serve as the data sets used for the traffic prediction experiments described in the next chapter.

In the last step, the experiment results are evaluated. During this master thesis, this mainly meant to plot the results from the *ITGSend* and *ITGRecv* processes to evaluate if the traffic generation was successful. Every host automatically logs its sent and received packets. After the traffic generation is finished, the experiment framework uses the *ITGDec* process to create plots out of the log files on the hosts. Afterwards, the plots are collected and stored at the control center node, so that they can be uploaded to a repository. Moreover, the experiment framework can also collect information provided by MaxiNet which illustrate the resource utilization of the MaxiNet Workers during the experiment runtime.

6 Traffic Prediction Experiments

In the following chapter, the conducted experiments in which the GKKF was used to predict traffic flows are presented. Starting with a description of the utilized traffic data set and an explanation why a certain subset of the contained flows was selected for forming the training and test sets. The second section shows how the data sets are preprocessed so that they can be used with the GKKF algorithm. The results of the experiments are then presented in the last section.

6.1 Experimental Data

The data set used during the experiments was collected as part of a study conducted by Benson et al. that aimed at analyzing the characteristics of network traffic in data centers [76]. During their study, Simple Network Management Protocol (SNMP) data, topology information and packet traces were collected from university, private enterprise and commercial cloud data centers. The collection of the packet traces was done by installing a sniffing tool at one or multiple switches in each network that logged all traffic routed through the switch during a time span of 12 hours over multiple days. A fraction of the gathered packet traces of two university data centers was made publicly available by the authors [77]. During this master thesis, the packet traces denoted as *UNI1* were used which contain 65 minutes worth of traffic data collected at an edge switch in a university data center consisting of 22 network devices and 500 servers. The decision was made for this data set because the traffic data is provided as a *.pcap* file. In comparison to text files, *.pcap* files are more convenient when it comes to the generation of statistics on the data and the extraction of single flows from it. For analyzing the traffic data, the terminal-based version of Wireshark called t-shark [78] was used. Another advantage of the selected data set is that the traces were collected in the year 2009 which makes them considerably more current than e.g. the data sets used in [25] or [22] which date back to the mid-nineties or eighties. Selecting relatively new data is important because the traffic characteristics of data centers in general could have changed completely over the years and thus, insights from the experiments would not be transferable to today's data centers.

In the first step, the packet traces were analyzed by gathering statistical information about the composition of the traffic data. For 87% of the traffic transmitted by the observed network switch, TCP was used as the transport layer protocol, only 13% was transmitted over UDP. A closer look on the TCP traffic then revealed which protocols were used at the application layer. The results illustrated in Figure 6.1 show that around 90% of the overall traffic uses one of the listed protocols, whereas Hypertext Transfer Protocol (HTTP) traffic makes up the largest share with 75%. Since the goal of this master thesis was to investigate if the progression of single flows can be predicted, we focused on traffic using the same application layer protocol because we initially assumed that those flows would already share enough flow characteristics so that knowledge about some of the flows would allow us to predict other unseen ones. The decision was made in favor of HTTP traffic because this group contains 65% of all the collected traffic data. However, further analysis of the HTTP traffic revealed that the single HTTP flows are very heterogeneous. The packet traces contain roughly 130 000 unique HTTP flows, whereas one flow represents a socket-to-socket connection between two hosts in the data center that

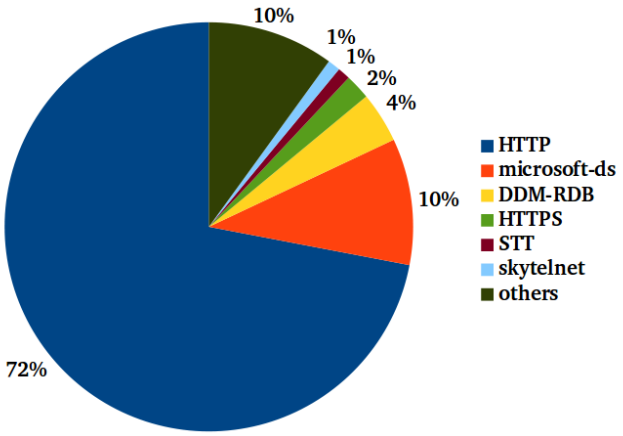
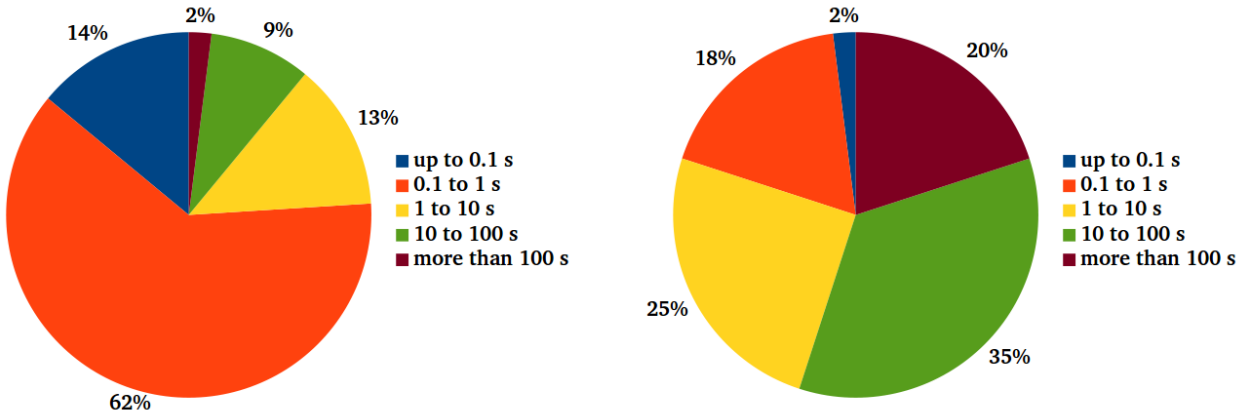


Figure 6.1.: Application Layer Protocol Shares of TCP Traffic



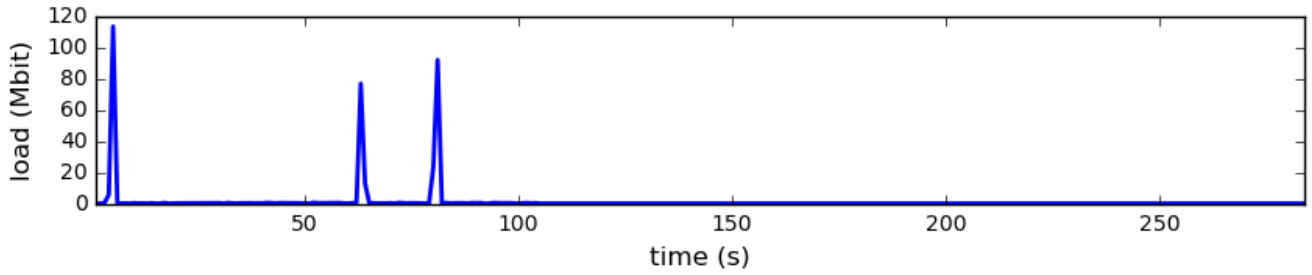
(a) Duration of HTTP flows

(b) Traffic Load Shares of Duration Groups

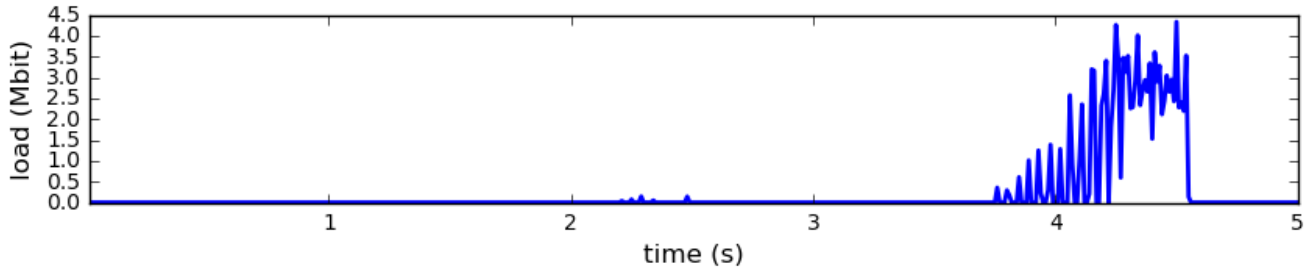
Figure 6.2.: Characteristics of HTTP Flows

starts with a TCP handshake and ends with a TCP teardown. As shown in Figure 6.2a, the durations among HTTP flows is highly diverse, whereas most of the flows are very short with a length between 0.1 and 1 second, as indicated by the red slice of the pie chart. To fully grasp the characteristics of HTTP flows it is important to know which share of the traffic load is caused by which duration group. This information is given in Figure 6.2b and together with Figure 6.2a it allows us to gain some interesting insights. The extremely short flows with a duration of up to 0.1 seconds cause only 2% of the traffic and can therefore be neglected. The same accounts for many flows in the duration group 0.1 to 1 second, though not for all of them. The most important group seems to be the one which contains flows with a rather short length between 1 and 100 seconds, since only 22% of the flows possess such a length and still they cause 60% of the traffic. All of these traffic flows could not be considered when aggregating the flows in a way as it was done in the related work described in Section 2.2 since the flows would only be represented by one or a few data points. Just the flows longer than 100 seconds could be represented sufficiently. As Figure 6.2b indicates, this duration group is responsible for 20% of the traffic. However, when examining the progression of high-volume long-living flows a typical flow pattern is observed. One example flow is shown in Figure 6.3a which clearly reveals the extremely bursty nature of the network traffic. Even though an already rather low sampling interval of 1 second is used, no flow structure can be observed since the short but very high peaks are only represented by one or two data points. For most of the time, the kbit values of the flow are either zero or have an insignificant value. This shows that even the long flows often just consist of short traffic peaks with long low-volume phases between them. For a TE system that tries to predict and reroute traffic in a network this means that it needs to be able to predict the flows on a small time scale because most of the traffic is transmitted in short high-volume traffic peaks. This also implies that the traffic data must be represented using a small sampling interval. Figure 6.3b shows the first 5 seconds of the same flow, however this time, a sampling interval of 0.01 seconds was used and now much more information about the peak structure can be obtained. As we can see, a peak itself consists of shorter peaks which from now on are called peak impulses. At the lowest possible sampling interval these impulses would represent single packets of the flows. In the vast majority of flows, the peaks consist of a rising phase in which the peak impulses gradually increase up to a certain maximum kbit value and a peak body where the traffic load rather remains constant. In the context of network congestion the rising phase is the most interesting part of the flow because here the traffic pattern of the flow is changing rapidly which causes the congestion. Therefore, the goal of a TE system should be a successful prediction of the course of the peak rise given only a small part of it.

Because of the high number of traffic flows, only elephant HTTP flows were extracted from the packet traces by selecting all flows that transmit at least 0.5% of the load transmitted by the biggest flow. For the extraction t-shark was used. This left us with a total of 603 single flows that transmit around 46% of all the HTTP traffic. The network experiment framework described in Chapter 5 could now be used to replay the flows in a emulated SDN and create files that would contain a time series of kbit values for every replayed flow. However, since the process of collecting the statistical information given as counters from all switches takes time, kbit values could only be produced at a sampling interval of at least 0.1 seconds. The described analysis of the single HTTP flows however revealed that because of the short peak duration a sampling interval of 0.01 seconds is needed to display enough structure of the peaks. Therefore, the data sets which were used during the traffic prediction experiments consist of time series of kbit values which were computed from the flows' *.pcap* files at a sampling interval of 0.01 seconds with the help of t-shark. Nevertheless, since the sniffing tool that produced the *.pcap* files was installed on the switch it had only access to the same information of packet arrival times that are also used for setting the counters collected from the controller. Therefore, both approaches can be seen as



(a) Sampling Interval of 1 Second



(b) Sampling Interval of 0.01 Second

Figure 6.3.: Segment of a Single High-volume HTTP Flow

equivalent. Differences in the data sets would rather be caused by inaccuracies of the traffic generator which replays the flows.
First experiments were now conducted using randomly selected flows from the elephant flow subset as the training and test sets for the GKKF. However, with this approach it was not possible to predict the progression of most of the test flows. The reason for the bad results were found by analyzing all the elephant flows selected during one run. Even though they all represent HTTP traffic, the peak structure is still very heterogeneous among the flows. This is the reason why in general, the structure of one HTTP flow does not give us enough information to predict the peak progression of other independent HTTP flows. Nevertheless, some of the flows showed similar characteristics and by further investigation we found out that these flows always belonged to the same client-server connection. This means that during the traffic collection time span of 65 minutes, a client communicated multiple times with the same server in the data center, probably sending repetitive requests to one application running on the server. Figure 6.4 shows two HTTP flows that stem from a recurring connection that occurred seven times during the observed 65 minutes time period. The plotted flows are not equal in impulse height and peak frequency but they possess a similar overall structure. Therefore, the traffic flows that originated from recurring connections were selected from the elephant flow subset, ordered in groups and in further traffic prediction experiments utilized to test if the flow structure of single groups could be learned and used to predict unseen flows.

6.2 Preprocessing the Data

As shown in Figure 6.4, the flow structure is extremely bursty when observed at a small sampling interval. As a consequence, the times series of kbit values produced from the flow packet traces contain many zero-value phases and generally high gradients. This makes the course of the time series difficult to learn for any model and is the reason why the approaches described in Section 2.2 aggregate the flows on a temporal scale. However, as demonstrated this would prevent a TE system from predicting short traffic peaks. Therefore, another approach was followed during this master thesis. The main idea is that the time series are seen as a time-discrete signal sampled using a sampling interval T_S. Instead of learning with the signal in the time domain we transform it to the frequency domain by making use of the Fourier Transform (FT). All learning and prediction computation done with the GKKF is done in the frequency domain. In the end, the resulting predictions are brought back to the time domain so that a TE system would receive them as kbit values which it could then use to compute optimal paths for the flows and reroute them accordingly. For representing the signal in the frequency domain, it first has to be split into smaller overlapping chunks of a certain length w. The starting points of the single chunks at which they are cut out of the original signal are defined by the sampling interval T_C. Consequently, the parameters T_S, T_C and w decide in how many chunks the original signal is split into and how much the chunks overlap. Since the signal is discrete in time the Discrete Fourier Transform (DFT) needs to be used for

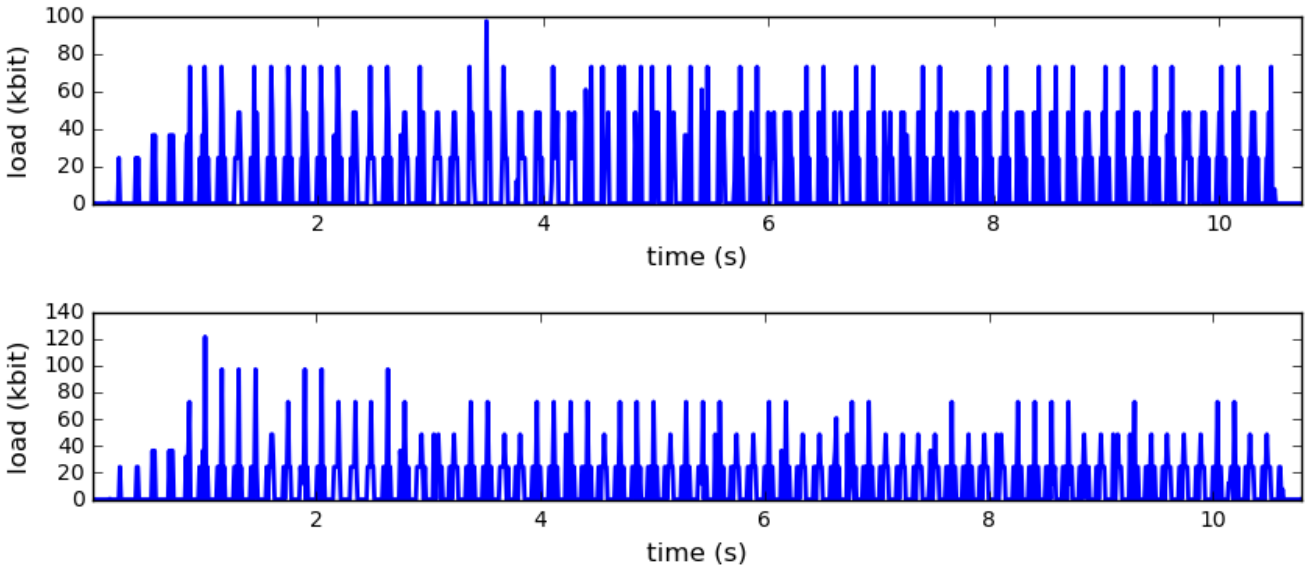


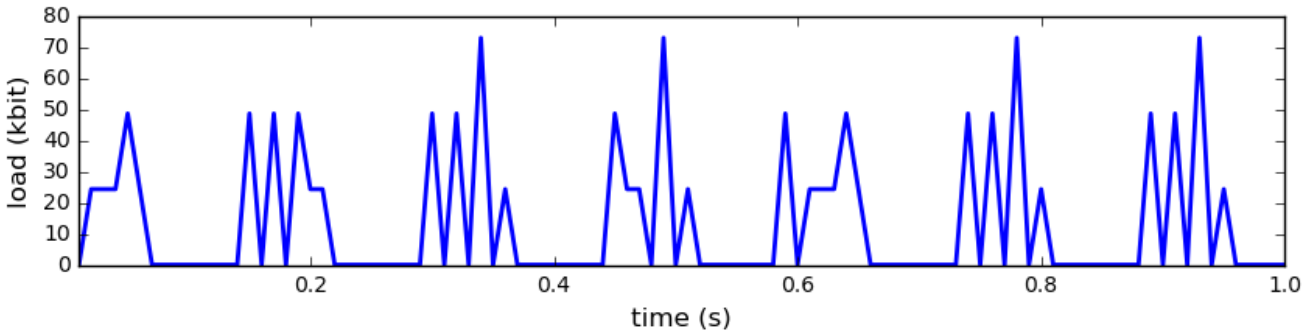
Figure 6.4.: Traffic Flows of Recurring Client-Server Communication

the transformation and by splitting the signal in chunks, basically a discrete-time Short-Time Fourier Transform (STFT) is performed, whereas a Fast Fourier Transform (FFT) algorithm is used for making the FT computations faster.

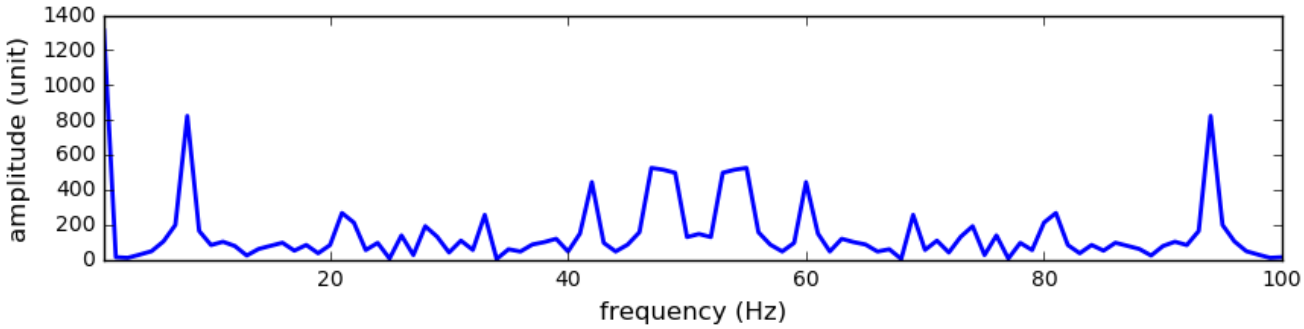
One main advantage of learning with the traffic signal in the frequency instead of the time domain is that the data sets the GKKF has to deal with become less bursty. Figure 6.5 compares the courses of different data series. In Figure 6.5a a 1 second long flow chunk is shown. When computing the FT of the chunk, a data series as shown in Figure 6.5b is produced. The data series consists of complex numbers that describe the amplitudes and phases of simple sine waves that can be combined to construct the original signal. In the plot the amplitudes of the needed sine waves are shown. Furthermore, the plot reveals that the frequency values get folded at the Nyquist frequency f_N which is half of the sampling frequency, $f_N = 0.5\, f_S = 0.5\, \frac{1}{T_S}$. For reconstructing the time signal from sine waves we therefore only need to save the first N values of the FT results, whereas $N = f_N + 1$. When transforming our results back to the time domain, the rest of the values can be reconstructed by inverting the imaginary parts of the saved complex numbers. However, in order to use the FT results with the GKKF, the complex numbers have to be split into their real and imaginary parts. The resulting data series consisting alternately of real and imaginary values is presented in Figure 6.5c which clearly shows that the data series is considerably less bursty then the original time series.
Another advantage of conducting the calculations in the frequency domain is that since the FT is built over a part of the signal, much more information about the flow structure is contained in every data point which makes it easier to observe the current condition of the flow. Furthermore, since $T_C > T_S$ the data set will contain less data points in the frequency than in the time domain. This is especially useful when the training set consists of very long flows that would otherwise result in a large number of data points. The selection of an appropriate value for T_C that leads to an overlap of the chunks also reduces artifacts in the frequency domain.

The influence of the transition to the frequency domain on the employed data set should now be illustrated using an example. We assume that the training set the GKKF should learn from consists of only one short high-volume flow with a duration of 10 seconds. When using a sampling interval $T_S = 0.01$ our training set $data_{train}$ contains a time series of kbit values observed at every 0.01 seconds and which possesses the dimensionality $data_{train} \in R^{1000x1}$. As mentioned before, the time signal is now split into single overlapping chunks. When using a sampling interval $T_C = 0.05$ and a chunk length of $w = 1$ the signal is split into 200 unique chunks that all contain 100 data points, whereas always 20 subsequent chunks overlap. In the next step, the FT is computed which returns a data series of 100 complex numbers for every chunk which represents a single observation in the frequency domain. As mentioned before, we only have to keep the first N frequencies but need to split them into their real and imaginary parts which in the end leaves us with a series of 102 data points for every chunk. Altogether, our training set will now have the dimensionality $data_{train} \in R^{200x102}$. As we can see, the ability to work with the traffic data needs to be paid with an increase of the observation dimensionality.

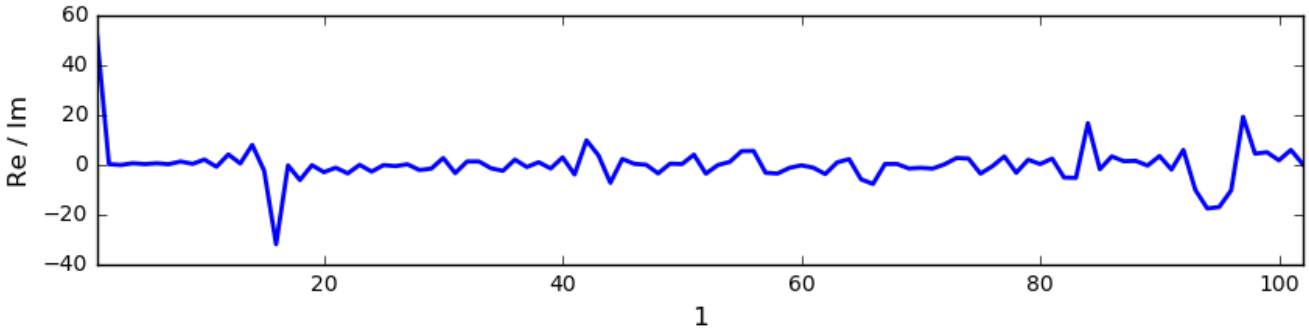
As shown in [39], the GKKF is able to process data with a very high dimensionality. Nevertheless, an attempt was made to reduce the dimensions needed for a single observation to lower their complexity and save computation time.



(a) Signal Chunk in the Time Domain



(b) Signal Chunk in the Frequency Domain



(c) Signal Chunk in the Frequency Domain with Split Complex Numbers

Figure 6.5.: Comparison of Data Series in Time and Frequency Domain

For that purpose a PCA was conducted on a data set after bringing it to the frequency domain and after standardizing it so that the set possesses zero mean and unit variance. The PCA uses an orthogonal transformation to compute Principal Components (PCs) corresponding to the dimensions of the data set that are ordered in a descending order, meaning the first PC explains more variance of the data set than any other PC. The presumption was that some frequencies might account for such a small amount of variance that their omission distorts the characteristics of the flow only insignificantly. It is important to mention that the PCA must always be conducted on the training set. The test set which is usually not fully available when testing also needs to be standardized and its dimensionality reduced. However, for the transformation, the PCs computed from the training set must be used. The number of dimensions by which the data sets can be reduced with a PCA differs between the groups of recurrent flows. Figure 6.6 shows the cumulated explained variance over the principal components for two separate flow groups. As the plot reveals, lesser dimensions are needed for flow group A than for flow group B to explain the same amount of variance. The variance values were obtained by selecting all flows except one flow of each group as separate training sets and performing a PCA on them. In a subsequent step, the unselected flows representing the test sets were reduced in their dimensionality using the computed PCs. Then the test sets were transformed back to the full dimensional space and brought back to the time domain to see how much information was lost through the dimensionality reduction. The influence of the reduction on the flow structure in the time domain when reducing the dimensionality from 102 to 80 dimensions is shown in Figure 6.7, whereas the blue lines represent the real appearances of the flows and the red lines their appearances after being transformed with the

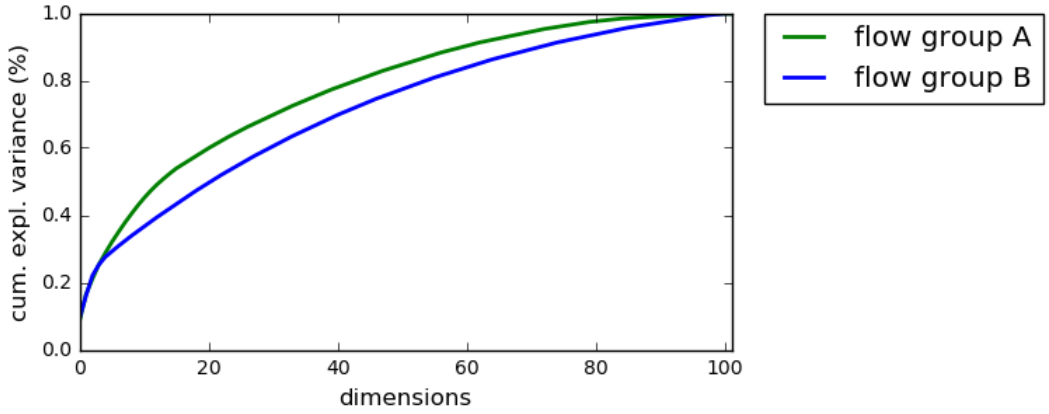


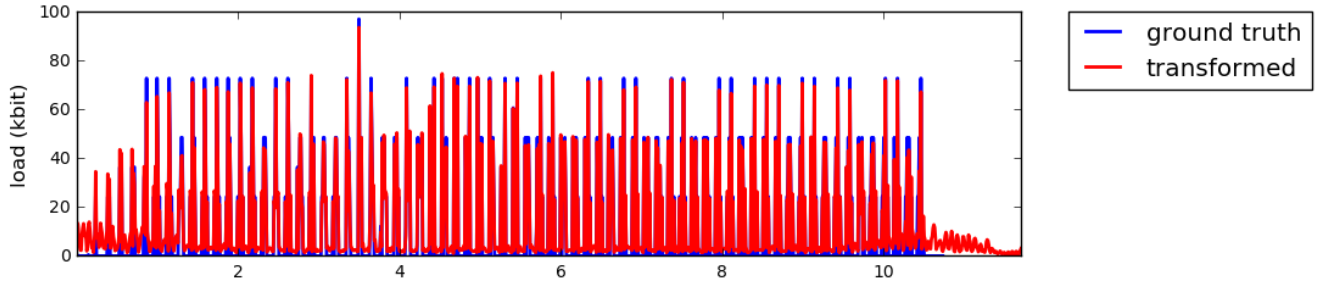
Figure 6.6.: Cumulative Explained Variance per Dimension

PCs. For flow group A, a reduction of 20 dimensions leaves us with 97.4% of explained variance. The loss of information is still acceptable, even though it is clearly visible in the plot. For flow group B, the remaining 80 dimensions are still enough to explain 93.3% of the variance. However, the plot demonstrates that for this flow group a larger percentage of information was contained in the omitted 20 dimensions since the heights of the flow peaks differ more from the original ones. Therefore, the results match the insights already obtained from Figure 6.6. Considering also the results for other flow groups, it has to be concluded that a PCA is only useful to a limited extend for reducing the dimensionality of the utilized traffic data since a reduction of only around 20 dimensions could be achieved in the best cases.

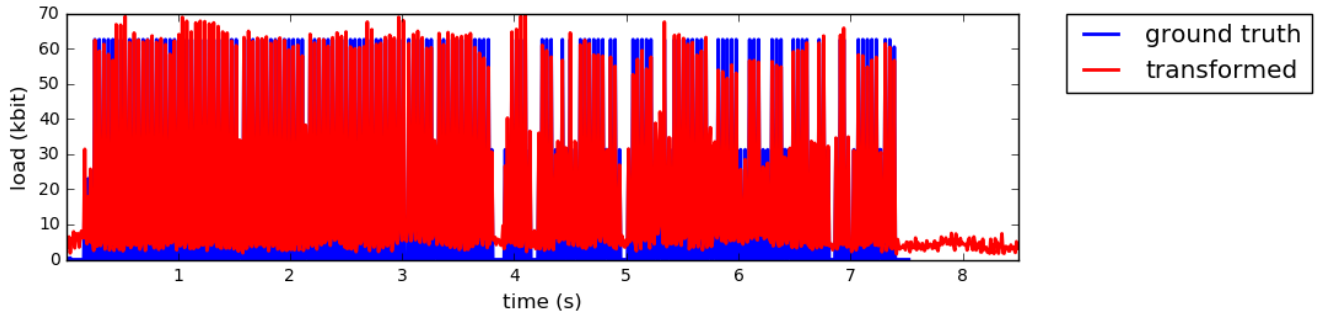
As mentioned before, the training sets used during the experiments contain observations of kbit values which were calculated from the flows' *.pcap* files. The kbit values the flows reach on the network are influenced by the applications running on the communicating hosts that produce the flows. Furthermore, their courses also depend on network elements like the network switches that forward the traffic or the links between them. All these components together form an underlying process or system, whereas the resulting kbit values of a single flow can be seen as observations emitted by the system. However, these observations do not represent the true state of the underlying process since different situations in the network defined by the condition of the flow-producing applications, switches, links or the appearance of other traffic, could produce the same kbit observations for the regarded flow. The true very complex state of the process is therefore hidden which is why the powerful concept of HMMs was used during this master thesis for modeling the process. Our training sets only contain the mentioned observations, however, as noted in Section 4.4, a window of observations can be used as an internal state representation. By computing the FT over chunks of the signal we are actually already building a window because the state is represented by the frequencies of a time signal chunk that consists of multiple observations. However, to extend the representation of the state during the prediction experiments, it was formed using the FTs of multiple chunks with increasing length. One possibility would e.g. be to use the FT of the next second and combine it with the FTs computed over the next 2 and 3 seconds. This allows us to include some more *long*-term behavior of the flow in the state representation. In such a scenario, an observation would be represented by the FT over the next second of the time signal.

6.3 Experimental Results

As mentioned before, the elephant flows were grouped by flows that stemmed from recurrent socket-to-socket connections. For the prediction experiments, the 20 groups with the highest numbers of flows were selected. The experiment results are shown in Table 6.1, whereas the column *# of flows* holds the information about how many flows each group contained. On average, one group consisted of 8 unique flows. The performance of the GKKF was tested on every flow group separately. During an experiment, always one flow was selected as the test set. The rest of the flows were used for constructing the training set. As noted in previous sections, the data set representing a flow consists of a time series of kbit values, whereas for the prediction experiments described in this section, a sampling interval $T_S = 0.01$ was used for producing the data series and a chunk sampling interval of $T_C = 0.05$ for splitting them into smaller chunks.
In Section 6.1 we showed the results of our traffic data analysis which revealed that HTTP traffic flows often consist of short traffic peaks that in turn can be divided in a rising phase and a phase with a mainly constant load level. Benson et al. [16] already demonstrated that a certain percentage of the overall traffic in a data center remains constant over a short period of time, meaning that the traffic load does not fluctuate by more than 20%. In such a scenario, the mean computed over the last short time period can be used for estimating the load of the flow in the next period. For this



(a) Test Flow from Flow Group A



(b) Test Flow from Flow Group B

Figure 6.7.: Influence of Dimensionality Reduction on the Flow Appearance

reason, the authors define such a traffic flow as predictable. However, at the time point from which on the flow load remains constant, congestion might already have happened in the network. Therefore, the more important parts of the flows are the peak rises, which is why we explicitly concentrate our predictions on these parts. We tested the ability of the GKKF to predict the peak rises after providing it with only a few observations from the beginning of the peak. The number of observations differs between the flows and ranges between 3 and 60 steps in the frequency domain which corresponds to an observed time interval of 0.15 to 3 seconds. When the observation stops and the true prediction begins, all flows still possess a low traffic load which then increases rapidly by around 150% on average during the next second. Therefore, when utilizing the definition from [16] all used traffic flows would need to be considered as unpredictable at the point of the last observation.
The GKKF was always used for predicting a time period of 1 second, resulting in 20 prediction steps in the frequency domain. The highest single impulse value predicted by the GKKF during this period was then compared to the true highest impulse value in the same period. The difference between both values divided by the true highest value represents the applied error metric for the prediction. The error should always be compared to the information that was already given in the observations. Therefore, another error metric was calculated for every flow group which helps to assess the quality of the prediction. In Table 6.1 the metric is shown in the *constant error* column. It was calculated like the prediction error only that the highest impulse value which emerged during the observation phase was used as the prediction, just as if we would *predict* a constant load. Since we only observe the beginning of the peak rise, the picked impulse value will always be smaller than the true one during the next second. Consequently, the true value will always be underestimated which is why all values in the column are negative.

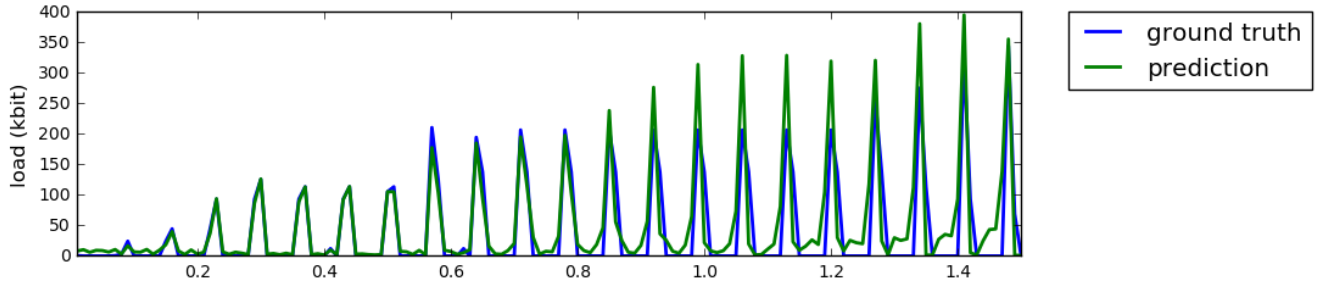
During the first set of experiments a chunk length $w = 1$ was used for all flow groups. The internal state representation was then built by combining the FTs of the next second, the next 2 seconds and the next 3 seconds of the signal. An observation was represented by the FT of the next second. The prediction errors obtained for all flow groups are shown in column *chunk len. = 1 s*. The performance of the GKKF for predicting a particular flow group is indicated by a red, yellow or green coloring of the associated table cell which corresponds to a bad, moderate or good prediction performance. Since it is difficult to assess the true prediction quality solely based on error metrics, a two-step approach was used.
In the first step, the computed prediction error and constant error values were utilized. Whenever the prediction error was above 50% or equally bad as the corresponding constant error value, the quality of the prediction was considered insufficient and the flow group marked in red. When the prediction error was lower than 50% and clearly smaller than the constant error, the group was marked in yellow. Only if a prediction error of less than 20% was achieved, the prediction was classified as *good* and marked in green. Subsequently, as a second step, a manual verification of the prediction was

				prediction error	
group ID	**# flows**	**constant error**	**optimal chunk len.**	**chunk len. = 1 s**	**chunk len. = optimal**
1	4	-69.7%	0.15 s	-43.1%	-2.0%
2	13	-60.1%	0.4 s	-36.3%	-8.4%
3	6	-63.5%	0.55 s	14.7%	5.6%
4	24	-57.6%	0.2 s	-62.3%	-22.8%
5	5	-55.9%	0.2 s	-53.1%	-12.1%
6	6	-46.9%	0.15 s	-52.8%	-13.2%
7	3	-60.0%	0.25 s	-4.5%	2.5%
8	10	-79.6%	0.3 s	-28.5%	-4.3%
9	4	-66.7%	1.95 s	-28.4%	-4.1%
10	4	-63.2%	0.5 s	-46.7%	-0.7%
11	5	-54.3%	0.25 s	-26.8%	-0.8%
12	18	-60.4%	0.95 s	-44.2%	-10.2%
13	5	-45.5%	0.55 s	-11.6%	-1.6%
14	3	-62.5%	0.35 s	-63.8%	-10.2%
15	11	-52.2%	0.25 s	-57.9%	-17.0%
16	13	-37.5%	0.55 s	-70.6%	-40.2%
17	3	-26.4%	0.95 s	-93.2%	-45.4%
18	4	-35.7%	0.45 s	151.9%	4.7%
19	7	-70.0%	0.15 s	-53.5%	-7.3%
20	3	-62.5%	0.95 s	-53.7%	2.0%

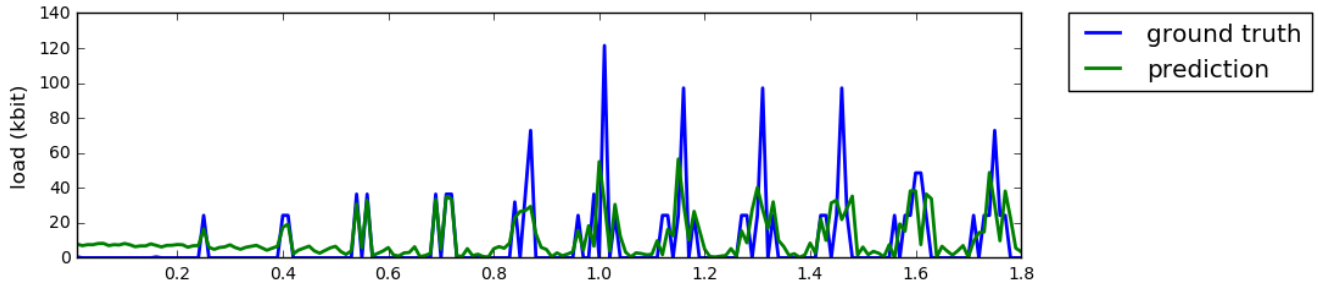
Table 6.1.: Prediction Experiments Results

done by comparing the predicted time series with its true counterpart. Only if the group dependent overall peak structure was visible in the predicted time series, the prediction of a flow group could retain its classification as *moderate* or *good*. If the prediction did not show the key characteristics found in the flows that made up the training set, the prediction quality got downgraded and the group marked in red.
When evaluating the prediction results for all groups using the described procedure, only a bad prediction performance was achieved for 11 out of 20 groups. A moderate prediction performance was obtained for 6 groups and a good one for 3 groups. To illustrate the differences between the performance classes, examples are shown in Figure 6.8. The plots always compare the predicted flow signal in the time domain to the true appearance of the flow, whereas the prediction corresponds to the mean prediction values obtained from the GKKF. The prediction variances were omitted in order to produce clearer plots. Figure 6.8a shows the prediction for flow group 3 which was classified as achieving a good performance. As clearly visible in the plot, the predicted flow has the same structure as the ground truth, even after time point 0.5 s from which on no further observations were obtained. The value of the highest impulse is slightly overestimated, resulting in a positive error prediction value of 14.7%. The results for flow group 19, shown in Figure 6.8b, again show matching flow structures for the predicted and true time series. However, this time, the predicted highest impulse value is further away from the true value which results in a prediction error of more than 50% and therefore, the group is marked in red. In Figure 6.8c we can see that for flow group 18 the highest impulse value is overestimated to a large degree. Furthermore, the predicted peak structure shows clear differences to the true appearance of the peak which is why classifying the prediction quality as *bad* is inevitable.

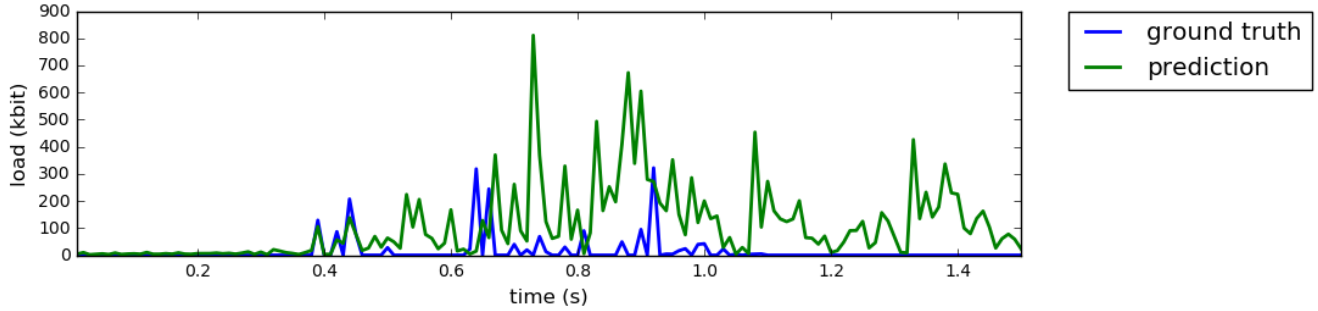
As mentioned, a chunk length of $w = 1$ was used so far for every flow group. However, the selected value might not be optimal for learning and predicting the peak structure of every flow group. When the test flow differs considerably



(a) Flow Group 3: Observations were obtained until time point 0.5 s



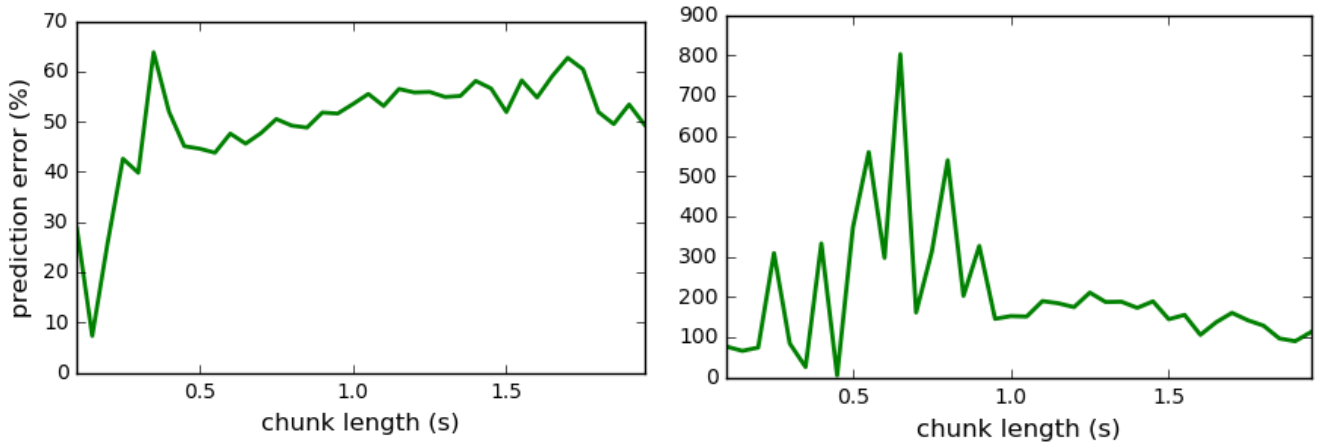
(b) Flow Group 19: Observations were obtained until time point 0.8 s



(c) Flow Group 18: Observations were obtained until time point 0.5 s

Figure 6.8.: Prediction Results with Chunk Length 1 Second

from the training flows in a very small part of the signal, it might be beneficial to use a smaller chunk length so that this *unique* part will be included into fewer chunks and therefore fewer observations. The assumption was verified by finding the optimal chunk length for every flow group, whereas values between 0.1 and 1.95 seconds for w were tested. The results are shown in column *optimal chunk len.*. The associated prediction errors achieved when utilizing these lengths are given in column *chunk len. = optimal*. By solely looking at the resulting error values the prediction quality seemed to improve for all flow groups. However, the low error values achieved for a specific chunk length could also just be a coincidence, meaning that the actual performance quality is bad but the highest predicted impulse is nevertheless close to the true value. For evaluating if a chunk length really improves the prediction, the error values achieved for all tested chunk lengths need to be analyzed. In Figure 6.9, the error values are shown for the flow groups 19 and 18. As we can see for flow group 19 in Figure 6.9a, the initially selected chunk length of 1 second was too long which is why a prediction error of more then 50% was achieved and the prediction performance therefore classified as *bad*. However, no matter which chunk length was selected, the highest impulse was always predicted with an error of not more then 63.8%. Furthermore, the error values do not oscillate. This indicates that the peak structure can be predicted in general and the highest impulse in particular when the optimal chunk length is selected. A different situation is observed for flow group 18. Even though the prediction error is low for the best chunk length, Figure 6.9b shows that the error values oscillate to a high degree in the area from 0.1 to 1 second. Moreover, when using a higher chunk length, no prediction error value below 100% can be achieved which leads to the conclusion that the peak structure cannot be correctly predicted.
For evaluating the prediction performance the described two-step approach was used again. Additionally, in a third step, the information obtained from plots similar to those in Figure 6.9 were utilized to possibly downgrade the performance level, marking the flow group in red. After the classification process, a good performance quality could be validated for 14 out of 20 groups. By selecting a better chunk length, the prediction of all flow groups that were earlier marked in

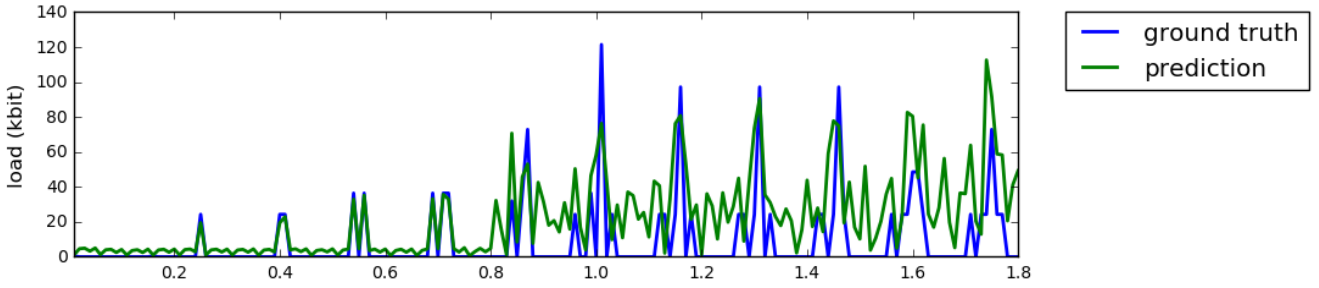


(a) Flow Group 19

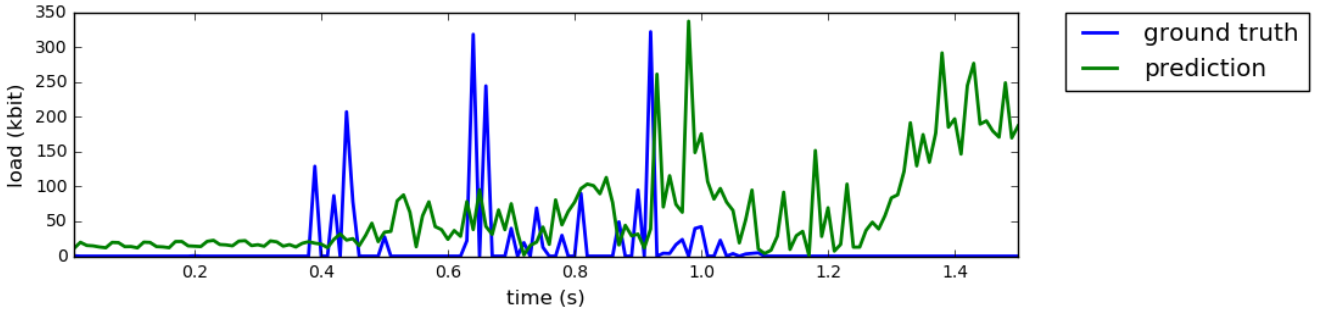
(b) Flow Group 18

Figure 6.9.: Chunk Length Dependent Prediction Errors

yellow or green could be improved. The reason is that for these groups, a correct prediction of the peaks' key characteristics could already be achieved when using a chunk length of 1 second. The same applies to the flow groups 5, 6, 14, 15 and 19 which were only marked in red because the error values exceeded the 50% mark. The prediction improvements achieved for flow group 19 are shown in Figure 6.10a. The key characteristics of the peak structure still get predicted and furthermore, the predicted highest impulse is closer to the true highest value. The fact that the highest impulse is predicted at a different time point is caused by the differences between the training and test flows.
For the remaining groups 4, 9, 16, 17, 18 and 20, only a bad performance could be achieved. Even though all flows forming one group stemmed from the same recurring socket-to-socket connection, the similarities between them was not strong enough so that the GKKF could learn the key characteristics from the training flows and use them to correctly predict the peak structure in the test flows. The prediction results for flow group 18, when using the optimized chunk length, are shown in Figure 6.10b. Despite predicting the true highest impulse value, the structure of the predicted time series is clearly different to the true one.



(a) Flow Group 19: Observations were obtained until time point 0.8 s



(b) Flow Group 18: Observations were obtained until time point 0.5 s

Figure 6.10.: Prediction Results with Optimized Chunk Length

7 Conclusion

During this master thesis, a framework was developed which allows the user to set up a scalable virtualized SDN with a custom topology and perform experiments with it. In one implemented type of experiment, statistical information was collected from the network devices and processed by the SDN controller to form observations that were planned to be used for the traffic prediction experiments. However, the analysis of real data center traffic revealed that a very small observation sampling interval would be needed, which cannot be produced with the currently implemented approach. Further development of the experiment framework should therefore focus on delegating some of the processing steps to the network devices, since observations could then be produced at a much higher sampling frequency. This could be achieved by extending the existing OVS implementation as shown in [79].
Performing the data processing partially decentralized at the network devices should also be considered in the development of any prospective TE system that aims at predicting and rerouting network traffic in SDNs. In such a system, network devices could use their own statistical information to identify elephant flows and produce observation sets which could then be fed into machine learning methods like the GKKF that would run locally on the devices and cope with the learning and prediction problem. The prediction results would then be sent to the controller and since this would only be done for elephant flows, the amount of produced control traffic would be drastically reduced. Furthermore, by splitting the prediction problem among the network devices, each instance of the utilized machine learning algorithm would have to deal with a smaller number of flows than one central algorithm. In a subsequent step, the central controller could form a global view of the current network situation from the received predictions and compute optimal paths for the dedicated flows, e.g. by formulating the goal of network throughput maximization as a multi-commodity flow problem, as shown by Agarwal et al. [11]. As a last step, the controller would then implement the optimal paths in the network in order to alter its routing behavior, thereby preventing upcoming congestion situations.

The approach used during this master thesis for predicting traffic flows focused on the small time scale prediction of peak structures of single flows because, as shown during the analysis of real network traffic, this is the only type of prediction that truly considers the existing bursty nature of the network traffic that is one important cause of network congestion. In the conducted traffic prediction experiments, the GKKF was used to predict the peak rises in flow groups consisting of single flows from recurrent socket-to-socket connections, whereas observations were only given to the GKKF whilst the flows' traffic load was still low. In 70% of the observed groups, a prediction of the highest traffic impulse during the next second could be achieved with an error of less then 20%. However, some constraints of these results need to be mentioned.
Firstly, a correctly predicted highest impulse value only states that no higher kbit value will be reached during a time interval of 10 ms in the next second. It does not directly tell us how high the overall load of the flow will be during this time span. However, when analyzing the peak structure of flows it can be observed that the impulses themselves often appear at a constant frequency depending on the associated flow group. Therefore, the total load produced during the next second could be estimated by deriving the impulse frequency from the observation phase and combining in with the prediction of the highest impulse value. Future work could evaluate, if it is possible to achieve a reasonable estimation of the total flow volume with this approach.
Secondly, the predicted flows were closely related since they stemmed from recurrent socket-to-socket connections. All of the flows occurred during a time span of only 65 minutes. Further research could now try to evaluate, if the similarities are also persistent over a longer period of time which would give hints at which frequency the model of the underlying system would have to be relearned. The observation of longer time periods could also reveal that the prediction of a certain flow group gets easier when more unique flow instances of one group are collected. Moreover, further research should assess, if recurrent flows make up a share of the total network traffic that is big enough so that a TE system utilizing the predictions could influence an adequate amount of the overall network traffic.
Thirdly, during this master thesis, the different flow groups were learned and tested independently to see, if the peak structures could generally be learned and predicted. However in practice, a TE system would need to handle a large number of flows which is why all or multiple flow groups probably would need to be combined to larger training sets so that only one or a few different models would have to be learned. Therefore, future research on the feasibility of a TE system capable of traffic prediction would need to investigate, if a prediction is still possible in such a scenario using the proposed approach.

Bibliography

[1] "Software-defined networking: The new norm for networks (white paper)." https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, 2012. [Online; accessed 23-Sept.-2016].

[2] A. Botta, W. de Donato, A. Dainotti, S. Avallone, and A. Pescapé, "D-itg 2.8.1 manual." http://traffic.comics.unina.it/software/ITG/manual/, 2013. [Online; accessed 23-Sept.-2016].

[3] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pp. 117–130, 2015.

[4] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pp. 19:1–19:6, 2010.

[5] P. Wette, M. Dräxler, and A. Schwabe, "Maxinet: Distributed emulation of software-defined networks," in *Networking Conference, 2014 IFIP*, pp. 1–9, 2014.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 92–99, 2010.

[7] M. Chiesa, G. Kindler, and M. Schapira, "Traffic engineering with equal-cost-multipath: An algorithmic perspective," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pp. 1590–1598, 2014.

[8] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, "Traffic engineering in software-defined networking: Measurement and management," *IEEE Access*, vol. 4, pp. 3246–3256, 2016.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pp. 3–14, 2013.

[10] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pp. 15–26, 2013.

[11] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks," in *INFOCOM, 2013 Proceedings IEEE*, pp. 2211–2219, 2013.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pp. 19–19, 2010.

[13] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM, 2011 Proceedings IEEE*, pp. 1629–1637, 2011.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 254–265, 2011.

[15] F. François and E. Gelenbe, "Towards a cognitive routing engine for software defined networks," *CoRR*, vol. abs/1602.00487, 2016.

[16] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pp. 8:1–8:12, 2011.

[17] P. K. Hoong, I. K. T. Tan, and C. Y. Keong, "Bittorrent network traffic forecasting with ARMA," *CoRR*, vol. abs/1208.1896, 2012.

[18] N. K. Hoong, P. K. Hoong, I. K. T. Tan, N. Muthuvelu, and L. C. Seng, "Impact of utilizing forecasted network traffic for data transfers," in *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pp. 1199–1204, 2011.

[19] Y. Yu, M. Song, Y. Fu, and J. Song, "Traffic prediction in 3g mobile networks based on multifractal exploration," *Tsinghua Science and Technology*, vol. 18, no. 4, pp. 398–405, 2013.

[20] B. Vujicic, H. Chen, and L. Trajkovic, "Prediction of traffic in a public safety network," in *2006 IEEE International Symposium on Circuits and Systems*, pp. 4 pp.–, 2006.

[21] N. C. Anand, C. Scoglio, and B. Natarajan, "Garch - non-linear time series model for traffic modeling and prediction," in *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, pp. 694–697, 2008.

[22] D. C. Park and D. M. Woo, "Prediction of network traffic using dynamic bilinear recurrent neural network," in *2009 Fifth International Conference on Natural Computation*, vol. 2, pp. 419–423, 2009.

[23] S. Chabaa, A. Zeroual, and J. Antari, "Identification and prediction of internet traffic using artificial neural networks," *Intelligent Learning Systems and Applications*, vol. 2, no. 3, pp. 147–155, 2010.

[24] P. Cortez, M. Rio, M. Rocha, and P. Sousa, "Internet traffic forecasting using neural networks," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pp. 2635–2642, 2006.

[25] Y. Chen, B. Yang, and Q. Meng, "Small-time scale network traffic prediction based on flexible neural tree," *Applied Soft Computing*, vol. 12, no. 1, pp. 274 – 279, 2012.

[26] C. N. Babu and B. E. Reddy, "Performance comparison of four new arima-ann prediction models on internet traffic data," *Journal of Telecommunications and Information Technology*, pp. 67–75, 2015.

[27] S. Chabaa, A. Zeroual, and J. Antari, "Anfis method for forecasting internet traffic time series," in *2009 Mediterrannean Microwave Symposium (MMS)*, pp. 1–4, 2009.

[28] Y. Liang and L. Qiu, "Network traffic prediction based on svr improved by chaos theory and ant colony optimization," *International Journal of Future Generation Communication and Networking*, vol. 8, no. 1, pp. 69–78, 2015.

[29] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[30] B. A. McElhoe, "An assessment of the navigation and course corrections for a manned flyby of mars or venus," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-2, no. 4, pp. 613–623, 1966.

[31] E. A. Wan and R. V. D. Merwe, "The unscented kalman filter for nonlinear estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pp. 153–158, 2000.

[32] D. J. Hsu, S. M. Kakade, and T. Zhang, "A spectral algorithm for learning hidden markov models," *CoRR*, vol. abs/0811.4413, 2008.

[33] H. Jaeger, "Observable operator models for discrete stochastic time series," *Neural Comput.*, vol. 12, no. 6, pp. 1371–1398, 2000.

[34] S. Vempala and G. Wang, "A spectral algorithm for learning mixture models," *J. Comput. Syst. Sci.*, vol. 68, no. 4, pp. 841–860, 2004.

[35] L. Song, B. Boots, S. M. Siddiqi, G. J. Gordon, and A. J. Smola, "Hilbert space embeddings of hidden markov models," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 991–998, 2010.

[36] G. Gebhardt, "Embedding kalman filters into reproducing kernel hilbert spaces," 2014.

[37] L. Ralaivola and F. d'Alche Buc, "Time series filtering, smoothing and learning using the kernel kalman filter," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 3, pp. 1449–1454, 2005.

[38] P. Zhu, B. Chen, and J. C. Prı´ncipe, "Learning nonlinear generative models of time series with a kalman filter in rkhs," *IEEE Transactions on Signal Processing*, vol. 62, no. 1, pp. 141–155, 2014.

[39] G. Gebhardt and G. Neumann, "The generalized kernel kalman filter - learning forward models from high-dimensional partial observations (internal report)," 2015.

[40] https://www.opennetworking.org/index.php. [Online; accessed 23-Sept.-2016].

[41] "Openflow switch specification version 1.5.1." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf, 2015. [Online; accessed 23-Sept.-2016].

[42] "The benefits of multiple flow tables and ttps." https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_Multiple_Flow_Tables_and_TTPs.pdf, 2015. [Online; accessed 23-Sept.-2016].

[43] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Hausder, "Hidden markov models in computational biology applications to protein modeling," *Journal of molecular biology*, vol. 235, no. 5, pp. 1501–1531, 1994.

[44] M. Gales and S. Young, "The application of hidden markov models in speech recognition," *Foundations and Trends in Signal Processing*, vol. 1, no. 3, pp. 195–304, 2007.

[45] A. D. Wilson and A. F. Bobick, "Parametric hidden markov models for gesture recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 9, pp. 884–900, 1999.

[46] C. Kohlschein, "An introduction to hidden markov models." http://www.tcs.rwth-aachen.de/lehre/PRICS/WS2006/kohlschein.pdf, 2006. [Online; accessed 23-Sept.-2016].

[47] Z. Ghahramani, "An introduction to hidden markov models and bayesian networks," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 15, no. 1, pp. 9–42, 2001.

[48] J. A. Bilmes, "A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models." http://melodi.ee.washington.edu/people/bilmes/mypapers/em.pdf, 1998. [Online; accessed 23-Sept.-2016].

[49] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[50] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[51] L. Wehbe, "Kernel properties - convexity." http://alex.smola.org/teaching/cmu2013-10-701x/slides/recitation_3_kernel_properties_convexity.pdf, 2013. [Online; accessed 23-Sept.-2016].

[52] N. Aronszajn, "Theory of reproducing kernels," *Transactions of the American Mathematical Society*, vol. 68, no. 3, pp. 337–404, 1950.

[53] A. Smola, A. Gretton, L. Song, and B. Schölkopf, "A hilbert space embedding for distributions," in *Proceedings of the 18th International Conference on Algorithmic Learning Theory*, ALT '07, pp. 13–31, 2007.

[54] L. Song, K. Fukumizu, and A. Gretton, "Kernel embeddings of conditional distributions: A unified kernel framework for nonparametric inference in graphical models," *IEEE Signal Processing Magazine*, vol. 30, no. 4, pp. 98–111, 2013.

[55] G. Gebhardt, A. Kupcsik, and G. Neumann, "Learning subspace conditional embedding operators (internal report)," 2015.

[56] M. Shannon, H. Zen, and W. Byrne, "Autoregressive models for statistical parametric speech synthesis," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 3, pp. 587–597, 2013.

[57] W. Feng, A. Goel, A. Bezzaz, W. Feng, and J. Walpole, "Tcpivo: A high-performance packet replay engine," in *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, MoMeTools '03, pp. 57–64, 2003.

[58] "Tcpreplay." http://tcpreplay.synfin.net/. [Online; accessed 23-Sept.-2016].

[59] S. Hong and S. F. Wu, "On interactive internet traffic replay," in *Recent Advances in Intrusion Detection: 8th International Symposium, RAID 2005*, pp. 247–264, 2006.

[60] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Comput. Netw.*, vol. 56, no. 15, pp. 3531–3547, 2012.

[61] "Kernel virtual machine." http://www.linux-kvm.org/. [Online; accessed 23-Sept.-2016].

[62] "Xen project." https://www.xenproject.org. [Online; accessed 23-Sept.-2016].

[63] "Virtual box." https://www.virtualbox.org/. [Online; accessed 23-Sept.-2016].

[64] "Using open vswitch without kernel support." https://github.com/openvswitch/ovs/blob/master/INSTALL.userspace.md. [Online; accessed 23-Sept.-2016].

[65] "Estinet." http://www.estinet.com/. [Online; accessed 23-Sept.-2016].

[66] "ns-3." https://www.nsnam.org/. [Online; accessed 23-Sept.-2016].

[67] "Omnet++." https://omnetpp.org/. [Online; accessed 23-Sept.-2016].

[68] "Ryu." https://osrg.github.io/ryu/. [Online; accessed 23-Sept.-2016].

[69] "Pox." https://github.com/noxrepo/pox. [Online; accessed 23-Sept.-2016].

[70] "Project floodlight." http://www.projectfloodlight.org/floodlight/. [Online; accessed 23-Sept.-2016].

[71] G.Karypis and V. Kumar, "Unstructured graph partitioning and sparse matrix ordering system version 2.0 (technical report)." http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.376&rep=rep1&type=pdf, 1995. [Online; accessed 23-Sept.-2016].

[72] D. Schwerdel, D. Hock, D. Günther, B. Reuther, P. Müller, and P. Tran-Gia, "Tomato - a network experimentation tool," *7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2011)*, 2011.

[73] "Openvz." https://openvz.org/. [Online; accessed 23-Sept.-2016].

[74] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pp. 63–74, 2008.

[75] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To infinity and beyond: Time warped network emulation," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pp. 1–2, 2005.

[76] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pp. 267–280, 2010.

[77] "Data set for imc 2010 data center measurement." http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html. [Online; accessed 23-Sept.-2016].

[78] "Terminal-based wireshark." https://www.wireshark.org/docs/man-pages/tshark.html. [Online; accessed 23-Sept.-2016].

[79] P. Gorja and R. Kurapati, "Extending open vswitch to l4-l7 service aware openflow switch," in *Advance Computing Conference (IACC), 2014 IEEE International*, pp. 343–347, 2014.

[80] K. B. Petersen and M. S. Pedersen, "The matrix cookbook." https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf, 2012. [Online; accessed 23-Sept.-2016].

A Appendix

A.1 List of Counters in OpenFlow Version 1.5.1

Counter	Bits	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group		
Reference Count (flow entries)	32	*Optional*
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group Bucket		
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

A.2 Derivation of a Finite-dimensional Kalman Gain Matrix in Hilbert Space

$$\mathcal{Q}_t = \Sigma_{\varphi_t}^{-} C_{\phi|\varphi}^T (C_{\phi|\varphi} \Sigma_{\varphi_t}^{-} C_{\phi|\varphi}^T + \kappa \mathbf{I})^{-1} \tag{A.1}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \boldsymbol{\Upsilon}_{x'}^T \left(\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T\right)^T \left[\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'} S_t^{-} \boldsymbol{\Upsilon}_{x'}^T \left(\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T\right)^T + \kappa \mathbf{I}\right]^{-1} \tag{A.2}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \boldsymbol{\Upsilon}_{x'}^T \boldsymbol{\Upsilon}_x (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Phi}_y^T \left[\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'} S_t^{-} \boldsymbol{\Upsilon}_{x'}^T \boldsymbol{\Upsilon}_x (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Phi}_y^T + \kappa \mathbf{I}\right]^{-1} \tag{A.3}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \left((\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'}\right)^T \boldsymbol{\Phi}_y^T \left[\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'} S_t^{-} \left((\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \boldsymbol{\Upsilon}_x^T \boldsymbol{\Upsilon}_{x'}\right)^T \boldsymbol{\Phi}_y^T + \kappa \mathbf{I}\right]^{-1} \tag{A.4}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \left((\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}\right)^T \boldsymbol{\Phi}_y^T \left[\boldsymbol{\Phi}_y (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'} S_t^{-} \left((\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}\right)^T \boldsymbol{\Phi}_y^T + \kappa \mathbf{I}\right]^{-1} \tag{A.5}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \mathbf{O}^T \underbrace{\boldsymbol{\Phi}_y^T}_{\mathbf{A}} (\underbrace{\boldsymbol{\Phi}_y \mathbf{O} S_t^{-} \mathbf{O}^T}_{\mathbf{B}} \underbrace{\boldsymbol{\Phi}_y^T}_{\mathbf{A}} + \kappa \mathbf{I})^{-1}, \tag{A.6}$$

whereas $\mathbf{O} = (\mathbf{K}_{xx} + \lambda_O \mathbf{I}_m)^{-1} \mathbf{K}_{xx'}$. In the next step, the matrix identity $\mathbf{A}(\mathbf{B}\mathbf{A} + \mathbf{I})^{-1} = (\mathbf{A}\mathbf{B} + \mathbf{I})^{-1}\mathbf{A}$ from [80] is applied and we arrive at

$$\mathcal{Q}_t = \boldsymbol{\Upsilon}_{x'} S_t^{-} \mathbf{O}^T (\boldsymbol{\Phi}_y^T \boldsymbol{\Phi}_y \mathbf{O} S_t^{-} \mathbf{O}^T + \kappa \mathbf{I}_m)^{-1} \boldsymbol{\Phi}_y^T \tag{A.7}$$

$$= \boldsymbol{\Upsilon}_{x'} S_t^{-} \mathbf{O}^T (\mathbf{G}_{yy} \mathbf{O} S_t^{-} \mathbf{O}^T + \kappa \mathbf{I}_m)^{-1} \boldsymbol{\Phi}_y^T \tag{A.8}$$

$$= \boldsymbol{\Upsilon}_{x'} \mathbf{Q}_t \boldsymbol{\Phi}_y^T, \tag{A.9}$$

whereas $\mathbf{Q}_t = S_t^{-} \mathbf{O}^T (\mathbf{G}\mathbf{O} S_t^{-} \mathbf{O}^T + \kappa \mathbf{I}_m)^{-1}$ is the finite-dimensional Kalman gain matrix in Hilbert space and $\mathbf{G}_{yy} = \boldsymbol{\Phi}_y^T \boldsymbol{\Phi}_y$ the Gram matrix of the embedded observations which forms the observation model matrix $\mathbf{G}\mathbf{O}$ together with $\mathbf{O}$.

A.3 Derivation of a Finite-dimensional Sub-space Kalman Gain Matrix in Hilbert Space

$$\mathcal{Q}_t^S = \Sigma_{\varphi_t}^{-} C_{\phi|\varphi}^{ST} (C_{\phi|\varphi}^S \Sigma_{\varphi_t}^{-} C_{\phi|\varphi}^{ST} + \kappa \mathbf{I})^{-1} \tag{A.10}$$

$$= \Sigma_{\varphi_t}^{-} (\boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S \boldsymbol{\Gamma}_x^T)^T \left[\boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S \boldsymbol{\Gamma}_x^T \Sigma_{\varphi_t}^{-} (\boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S \boldsymbol{\Gamma}_x^T)^T + \kappa \mathbf{I}\right]^{-1} \tag{A.11}$$

$$= \Sigma_{\varphi_t}^{-} \boldsymbol{\Gamma}_x \mathbf{L}_O^S \underbrace{\overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T}_{\mathbf{A}} (\underbrace{\boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S \boldsymbol{\Gamma}_x^T \Sigma_{\varphi_t}^{-} \boldsymbol{\Gamma}_x \mathbf{L}_O^S}_{\mathbf{B}} \underbrace{\overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T}_{\mathbf{A}} + \kappa \mathbf{I})^{-1}, \tag{A.12}$$

whereas $\mathbf{L}_O^S = (\overline{\mathbf{K}}_{xx}^T \overline{\mathbf{K}}_{xx} + \lambda_O \mathbf{I}_n)^{-1}$. In the next step, the matrix identity $\mathbf{A}(\mathbf{B}\mathbf{A} + \mathbf{I})^{-1} = (\mathbf{A}\mathbf{B} + \mathbf{I})^{-1}\mathbf{A}$ from [80] is applied and we arrive at

$$\mathcal{Q}_t^S = \Sigma_{\varphi_t}^{-} \boldsymbol{\Gamma}_x \mathbf{L}_O^S (\overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T \boldsymbol{\Phi}_y \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S \boldsymbol{\Gamma}_x^T \Sigma_{\varphi_t}^{-} \boldsymbol{\Gamma}_x \mathbf{L}_O^S + \kappa \mathbf{I}_\mathbf{n})^{-1} \overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T \tag{A.13}$$

$$\boldsymbol{\Gamma}_x^T \mathcal{Q}_t^S = P_t^{-} \mathbf{L}_O^S (\overline{\mathbf{K}}_{xx}^T \mathbf{G}_{yy} \overline{\mathbf{K}}_{xx} \mathbf{L}_O^S P_t^{-} \mathbf{L}_O^S + \kappa \mathbf{I}_\mathbf{n})^{-1} \overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T \tag{A.14}$$

$$\boldsymbol{\Gamma}_x^T \mathcal{Q}_t^S = P_t^{-} \mathbf{L}_O^S (\overline{\mathbf{K}}_{xx}^T \mathbf{G}_{yy} \mathbf{O}^S P_t^{-} \mathbf{L}_O^S + \kappa \mathbf{I}_\mathbf{n})^{-1} \overline{\mathbf{K}}_{xx}^T \boldsymbol{\Phi}_y^T \tag{A.15}$$

$$\boldsymbol{\Gamma}_x^T \mathcal{Q}_t^S = \mathbf{Q}_t^S \boldsymbol{\Phi}_y^T, \tag{A.16}$$

whereas $\mathbf{G}\mathbf{O}^S = \mathbf{G}_{yy}\mathbf{O}^S$ is the observation model matrix and $\mathbf{Q}_t^S = P_t^{-} \mathbf{L}_O^S (\overline{\mathbf{K}}_{xx}^T \mathbf{G}\mathbf{O}^S, P_t^{-} \mathbf{L}_O^S + \kappa \mathbf{I}_\mathbf{n})^{-1}$ the finite-dimensional sub-space Kalman gain matrix in Hilbert space.